UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL
DEPARTAMENTO DE ESTRUTURAS

Adaptatividade *hp* aplicada em malhas de elementos finitos

Autor: **Edimar Cesar Rylo**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL
DEPARTAMENTO DE ESTRUTURAS

Adaptatividade *hp* aplicada em malhas de elementos finitos

Autor: **Edimar Cesar Rylo**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

Curso: Engenharia Civil
Área de concentração: Estruturas

Dissertação de Mestrado apresentada à Comissão de Pós Graduação da Faculade de Engenharia Civil, como requisito para obtensão do título Mestre em Engenheria Civil.

Campinas, Agosto de 2002
S. P. - Brasil

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

R985a

Rylo, Edimar Cesar
Adaptatividade hp aplicada em malhas de elementos finitos / Edimar Cesar Rylo.--Campinas, SP: [s.n.], 2002.

Orientador: Philippe Remy Bernard Devloo.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Civil.

1. Método dos elementos finitos. 2. Análise numérica 3. Analise de erros (Matemática). 4. Método dos elementos finitos – Processamento de dados. I. Devloo, Philippe Remy Bernard. II. Universidade Estadual de Campinas. Faculdade de Engenharia Civil. III. Título.

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA CIVIL
DEPARTAMENTO DE ESTRUTURAS

DISSERTAÇÃO DE MESTRADO

Adaptatividade *hp* aplicada em malhas de elementos finitos

Autor: **Edimar Cesar Rylo**
Orientador: **Prof. Dr. Philippe Remy Bernard Devloo**

Prof. Dr. Philippe Remy Bernard Devloo
instituição DES/FEC/UNICAMP

Prof. Dr. Francisco Antônio Menezes
instituição DES/FEC/UNICAMP

Profa. Dra. Sônia Maria Gomes
instituição DMA/IMECC/UNICAMP

Campinas, 12 de Agosto de 2002

Agradecimentos

Ao Departamento de Estruturas da Faculdade de Engenharia Civil - UNICAMP pela infraestrutura.

À FINEP, CNPQ e PETROBRAS pelo financiamento da infraestrura computacional.

À FAPESP pela bolsa de estudos e reserva técnica.

Ao amigo e orientador Prof . Dr. Philippe R. B. Devloo pela empolgante orientação e compartilhamento irrestrito de suas idéias.

Aos amigos e professores do departamento de estruturas pelo suporte e interesse no desenvolvimento do trabalho.

Aos amigos do LabMeC em especial, ao Edivaldo pela rapadura, ao Gustavo pelas idéias e pela manutenção da rede do laboratório, Fábio por pilotar a churrasqueira, Erick pela diferença, Cedric pela qualidade de seu trabalho, Cafu pelo seu interesse e aos demais aqui não citados.

A família Garcia que me acolheu durante o desenvolvimento deste trabalho, para os quais tenho muita gratidão não apenas pela acolhida más pelo ambiente, pelas lições que aprendi e pelo apoio que sempre me foi dado.

À Deus.

Resumo

Uma estratégia *hp* adaptativa é implementada com base em um parâmetro de erro estimado localmente. A análise do padrão de refinamento em cada elemento é feito aresta a aresta, o que torna o método possível para qualquer tipo de elemento independente de sua dimensão.

Métodos *hp* adaptativos são desenvolvidos a mais de uma década [12], sendo a escolha do padrão de refinamento feita, inicialmente, com base no conhecimento da solução e de sua seqüência ótima de refinamento.

A estratégia proposta é implementada no ambiente de programação científica PZ [10]. As seguintes contribuições foram feitas ao ambiente PZ:

- Implementação de uma estratégia para identificação de elementos de referência para a partição da malha
- Implementação de uma estratégia para obtenção de *patches*, baseada nos elementos de referência
- Implementação de clones de um *patch* e através deste clone, o clone uniformemente refinado e processado
- Transferência de solução entre malhas. Um método para calcular a matriz de transferência entre malhas é implementada baseada em uma projeção L^2 do espaço de interpolação da malha grossa no espaço da malha fina.
- Uma classe para implementar a análise uni-dimensional de elementos, escolhendo o padrão *hp* ótimo para cada elemento.

Estas classes foram usadas para a implementação da estratégia *hp* adaptativa baseada na análise de arestas. Os resultados mostram que esta estratégia conduz a malhas com taxas de convergência exponencial, mesmo para problemas com singularidades.

Exemplos de validação são mostrados ao final do trabalho.

Abstract

An adaptive strategy is presented which uses an automatically generated local *patch* to estimate the error and implements a refinement criterium based on regularity analysis of one dimensional problems along the edges of the elements.

hp-Adaptive methods have been available for more than one decade [12]. Thus far, the choice of h, p or *hp*-refinement has been based on the a-priori knowledge of the regularity of the solution and their corresponding optimal sequence of refinement pattern [8].

The proposed strategy is implemented within the object oriented environment PZ [10] for the development of scientific software. The following capabilities of the PZ environment were either used or added :

- Implements a strategy to obtain reference elements for mesh partition
- Implements a strategy to obtain a *patch* based on one reference element
- Implements a *patch* clone an over this clone the uniformly refined *patch* is obtained an precessed.
- Transfer of solution between meshes. A method for computing a transfer matrix between meshes is implemented based on the L^2 projection of the interpolation space of the coarse mesh onto the fine mesh
- A class which implements a one dimensional optimal *hp* refinement analysis based on the comparaison of all possible refinement patterns.

These interfaces are used to implement and edge-based adaptive *hp*-refinement strategy. The results show that the adaptive strategy is able to produce hp refined meshes with exponential convergence rates, even for singular problems.

Several examples show the generality and aplicability of the strategy for different simulations.

Sumário

Capítulo 1

Introdução

Esse trabalho propõe o desenvolvimento e implementação de métodos para auto-adaptabilidade em malhas de elementos finitos bi e tri-dimensionais, a ser implementado no ambiente de programação de elementos finitos PZ, o qual já tem um grande número de métodos implementados pelo grupo de pesquisa do Orientador (ver [11, 4, 13, 9, 14, 15, 10]).

O método dos elementos finitos, assim como todo método numérico, necessita de uma discretização inicial do espaço de aproximação, sendo isto feito através de dois parâmetros básicos:

- a ordem polinomial do espaço de funções teste (parâmetro p);
- discretização do domínio onde será feita a aproximação por elementos finitos (parâmetros h);

No caso do método dos elementos finitos, demonstra-se que a qualidade da aproximação melhora com um refinamento uniforme do espaço de aproximações conforme proposição deste trabalho.

O contraponto é que o refinamento uniforme no domínio implica no aumento do número de graus de liberdade do problema.

A dimensão do sistema a ser resolvido está relacionada diretamente ao custo computacional envolvido no processo, podendo este custo ser entendido como a quantidade de memória utilizada, bem como o custo decorrido do tempo de utilização de equipamento computacional.

Assim, há uma relação qualidade da aproximação versus performance, cujo equacionamento representa um dos problemas práticos dos profissionais que se utilizam de ferramentas numéricas.

A auto-adaptabilidade, na forma aqui abordada, visa a obtenção de uma malha cuja relação erro de aproximação versus número de graus de liberdade é otimizada, de modo a indicar a malha de elementos finitos com discretização e espaço de funções teste cujo erro é mínimo para aquele número de graus de liberdade. Ainda que não seja um dos objetivos

iniciais deste projeto, a questão de performance é um parâmetro fundamental em qualquer problema de análise numérica e esta deverá ser priorizada em trabalhos futuros.

A metodologia utilizada consiste na estimação de um parâmetro representativo do erro da aproximação para indicar onde é necessário a adaptação da malha. Para a obtenção desse resultado são seguidos os seguintes passos:

1. Dada uma solução inicial e uma solução obtida de um espaço de aproximação refinado (parâmetros h e p), calcular a diferença entre estas duas soluções em cada elemento, sendo esse valor considerado a estimativa do erro;

2. Com o erro obtido determinar quais elementos apresentam maior erro e desta forma, identificar onde o refinamento é necessário;

3. A análise dos elementos é feita aresta a aresta, buscando dentre as possíveis combinações de refinamento desta aresta, com um número de graus de liberdade pré determinado, qual aquela que mais se aproxima da solução proveniente do refinamento uniforme hp da aresta. Esta metodologia foi apresentada em [8] e seus resultados mostraram eficazes para espaços unidimensionais, levando a uma taxa de convergência ótima para o parâmetro h e quase ótima para o parâmetro p.

4. Os parâmetros hp, obtidos anteriormente da análise unidimensional, são aplicados na malha original levando a malha adaptada.

Um aspecto a ser destacado nessa metodologia é o fato de se trabalhar com a comparação de dois espaços de aproximação hierárquicos, e desta forma independe do tipo de problema, do tipo de elemento utilizado, do número de dimensões etc.

Esta generalização do método é muito importante sob o aspecto de implementação, uma vez que o objetivo do trabalho é a implementação desta técnica em um ambiente de programação de elementos finitos orientado a objetos, o PZ [10].

Outro aspecto a ser destacado é o desenvolvimento de métodos para a estimação do erro da aproximação localmente.

Como mencionado anteriormente, a qualidade da aproximação é medida através da sua comparação com uma aproximação proveniente de uma malha com espaço de aproximação enriquecido.

No caso dos métodos utilizados como base para este desenvolvimento, o método estava inserido em uma metodologia *Multigrid*, onde a utilização de malhas refinadas e não refinadas é comum para a resolução do sistema de equações.

As principais contribuições deste trabalho são:

1. Implementação de classes para a subestruturação de malhas de elementos finitos. Durante a fase de estudo do ambiente de programação de elementos finitos PZ (ver [10]),

foi implementado um código de subestruturação de malhas. A utilização da subestruturação abre caminho para a implementação de métodos paralelos, pois os problemas de acoplamento de sub-domínios aos domínios é um dos problemas tratados pela subestruturação.

2. Estudo e documentação do código auto-adaptativo implementado em [8]. Também foi implementado uma classe matricial no ambiente PZ para tratar com matrizes do tipo bloco sobreposto. Esta matriz servirá de base para a implementação de um pré-condicionador baseado em blocos sobrepostos. Acredita-se que este pré-condicionador irá melhorar a performance do *solver multigrid*, no qual está implementada a metodologia auto-adaptativa estudada.

3. Implementação de uma metodologia para particionar um domínio, com base nos elementos particionados gerar um *patch* de elementos, sendo este *patch* utilizado como base para a criação de uma malha. Esta metodologia possibilita a implementação de um estimador de erros local.

4. Adaptação do código auto-adaptativo implementado em [8] para a utilização do estimador de erros local citado no ítem anterior. Esta abordagem mostrou que o tempo para a obtenção de uma malha adaptada é sensivelmente reduzido com a utilização do estimador de erros local.

5. Qualificação do código para exemplos que apresentam singularidades. Comparação dos resultados obtidos com o estimador de erro local com os resultados obtidos com o estimador global.

A organização do trabalho consiste da divisão do assunto em quatro conjuntos:

1. Revisão Bibliográfica, Método dos Elementos Finitos e Métodos Auto-adaptativos: consiste na descrição da teoria no qual o trabalho está envolvido, incluindo a descrição de metodologias e ferramentas que são utilizadas no desenvolvimento do trabalho. Isto é feito nos Capítulos 2,3 e 4;

2. Descrição da Metodologia Base: Capítulo 5, onde é feita a descrição dos aspectos teóricos e da implementação desta metodologia em um ambiente *multigrid.* Esta descrição detalhada faz-se necessária em função de, através da utilização de programação orientada a objetos, várias classes já implementadas estarem sendo utilizadas nestes trabalho;

3. Descrição da Metodologia Adotada: Implementação desta no Ambiente PZ e Testes de Validação: descrição da metodologia desenvolvida e implementada neste trabalho, sendo realizada no Capítulo 6;

4. Extensões e Conclusões: Realizada nos Capítulos 7 e 8, onde é realizada a análise do trabalho e de seus resultados, indicando as suas possíveis extensões.

Assuntos estudados durante o trabalho, cujos vínculos não são diretamente relacionados ao trabalho são apresentados na forma de anexos. Os anexos estão organizados da seguinte forma:

1. Estimador de erros de Zienkiewicz e Zhu [26]: dada a importância deste estimador de erros e a sua ampla utilização, este estimador de erros é descrito no Apêndice A;

2. Implementação de Matriz Bloco Sobreposto: de modo a melhorar a performance do *solver multigrid*, no código auto-adaptativo utilizado como base para o desenvolvimento do trabalho, foi implementada uma matriz composta por blocos sobrepostos. Esta matriz servirá de base para a implementação de um pré-condicionador do tipo bloco sobreposto, o qual deverá melhorar a performance deste código. Os conhecimentos utilizados nesta implementação são documentados no Apêndice B;

3. Reimplementação de Mudança de Coordenadas no PZ: aqui é descrita a primeira abordagem ao ambiente PZ, onde as funções de mudança de coordenadas foram reimplementadas, incluindo o sistema de coordenadas polar e o sistema cilíndrico ao PZ. Esta implementação é descrita no Apêndice C;

4. Subestruturação: a implementação de métodos de subestruturação no ambiente PZ serviu para o aluno se ambientar a alguns conjuntos de classes do PZ de grande importância ao desenvolvimento do trabalho, destacando-se as classes de malhas, elementos e matrizes. Os conceitos de estruturação das malhas e elementos e nós foram estudados. Os conceitos e a implementação do código são descritos no Apêndice D.

Capítulo 2

Revisão Bibliográfica

Todo trabalho de pesquisa necessita de uma revisão bibliográfica, de modo a mostrar o estado da arte no assunto de interesse, servindo também para verificar, dentre as diversas possibilidades de implementação, se resultados de outros trabalhos podem ser aproveitados neste.

Com a revisão bibliográfica, tem-se, além do ganho conceitual, uma melhor visão do caminho a seguir, em função das experiências relatadas por outros autores.

A revisão bibliográfica neste trabalho consistiu de :

- estudo da bibliografia específica aos assuntos relativos ao tema do trabalho e
- estudo de ferramentas necessárias ao desenvolvimento do trabalho.

A seguir são descritos os ítens constantes da revisão bibliográfica.

2.1 Revisão Bibliográfica Específica para o Desenvolvimento do Trabalho

Os principais tópicos necessários ao desenvolvimento do trabalho são, na seqüência:

1. Ferramentas Necessárias ao Desenvolvimento do Trabalho;
2. Método dos Elementos Finitos;
3. Estimadores de Erro;
4. Adaptabilidade e
5. Critérios para auto-adaptabilidade, baseados em análise de estimativas de erro.

A descrição destes tópicos, dada sua importância é feita em ítens à parte.

2.1.1 Ferramentas Necessárias ao Desenvolvimento do Trabalho

As principais ferramentas necessárias para o desenvolvimento deste projeto são *softwares*, destacando-se o sistema operacional *Linux*, os compiladores *GNU-gcc*, editores *Latex* e o *software* de visualização de resultados Open DX, todos de domínio público e com documentação disponível para estudo.

Além dos programas específicos, faz-se necessário destacar a utilização da filosofia de Orientação a Objetos e a UML (*Unified Modelling Language*), as quais são descritas abaixo.

Programação Orientada a Objetos

Essa filosofia de programação difere da programação procedural, comum em outros tipos de linguagens científicas tal como *pascal*, por seu comportamento não ser ditado pela seqüência do código e sim pelo comportamento dos objetos componentes do programa.

As principais vantagens da programação orientada a objetos estão relacionadas ao gerenciamento do código e à sua reutilização.

A programação orientada a objetos apresenta as seguintes características:

- encapsulamento;
- herança / derivação;
- polimorfismo.

O encapsulamento consiste em cada objeto apresentar uma série de dados e funções, cujo acesso é controlado, sendo somente permitido o acesso aos dados e funções públicas. Desse modo, acesso e modificação dos dados do objeto podem ser controlados sendo permitido apenas em determinadas funções, tornando o código assim mais seguro.

Com relação à herança e derivação, essa filosofia de programação permite que sejam criadas classes derivadas de outras já existentes, tendo as classes derivadas a herança de todas as características da classe mãe, podendo ser implementados apenas os métodos específicos e aqueles cujo comportamento na classe derivada é diferente do comportamento previsto na classe mãe. Assim, o trabalho de implementação é reduzido.

A linguagem de programação utilizada é C++, com bibliotecas ANSI dos pacotes Linux, de domínio público.

O orientador possuir um conjunto de bibliotecas de elementos finitos desenvolvidos em C++, ambiente esse utilizado como base para o desenvolvimento do trabalho. O aluno já possui experiência nessa linguagem, experiência essa adquirida em dois trabalhos de iniciação científica [23, 22].

UML - Unified Modeling Language

A UML consiste da tentativa de criar uma linguagem para modelar desde o planejamento até a execução do código.

Este tipo de representação do código é muito útil sob os seguintes aspectos:

- independe da linguagem de programação a utilizar;
- todo o código pode ser planejado através de uma série de diagramas que podem descrever o comportamento global de um código, os comportamentos de objetos e a implementação de métodos propriamente dita;
- induz ao desenvolvimento da documentação previamente à implementação do código, o que em grande parte dos casos conduz à identificação de problemas de implementação antes que estes ocorram;
- os diagramas UML representam o comportamento de programas orientados a objetos de melhor maneira que os fluxogramas tradicionais.

No caso deste trabalho, o estudo de UML foi superficial, sendo utilizados os diagramas de classes para a representação de códigos implementados. Os diagramas de uso, úteis ao planejamento de códigos, não foram estudados a fundo.

O não aprofundamento por parte do aluno nesta poderosa ferramenta se deve ao fato de estarmos, até o momento, implementando componentes em um ambiente já extenso e com planejamento já realizado pelo orientador, não existindo questões de planejamento global de *softwares*, o que ocorreria caso se procurasse desenvolver tudo sem a utilização de orientação a objetos.

As classes foram geradas diretamente através da implementação de códigos, sendo feitas análises do comportamento de objetos segundo parâmetros matemáticos, sendo para tal análise utilizado o *software Mathematica* da *Wolfram* (*Mathematica 4* © *Wolfram Research* - informações: *http://www.wolfram.com*).

2.1.2 Ambientação aos Trabalhos já Desenvolvidos

Como esse projeto está inserido dentro de uma linha de pesquisa [11], que já possui uma série de resultados implementados, os resultados desse trabalho deverão estender àqueles já apresentados, possuindo estrutura e documentação padrão, de tal modo que esse possa ser, futuramente, utilizado como base de implementação de outras técnicas.

Assim é fundamental a introdução do aluno ao ambiente, possibilitando o desenvolvimento do trabalho.

Essa etapa iniciou-se pelo estudo do ambiente PZ e, de modo a familiarizar o aluno com o ambiente. Como tarefa inicial implementou-se no ambiente PZ classes para transformação de coordenadas, sendo implementadas classes para coordenadas cilíndricas e cartesianas.

Também para a ambientação, todas as classes implementadas pelo aluno foram documentadas, utilizando um pacote específico, o *Doxygen* (Doxygen©1997-2002 *by* Dimitri Van Heesch), adotado na documentação do ambiente PZ. Desta forma, a documentação do código gerado tem o mesmo padrão de documentação do código já implementado.

Ambiente PZ

O ambiente PZ conta com grande tempo de desenvolvimento e possui um extenso número de funções.

O conhecimento da filosofia utilizada e dos métodos existentes é fundamental para o entendimento de como a técnica de refinamento deve ser implementada, que tipo de características deve possuir, quais dados de entrada deverão ser previstos e como apresentar os resultados.

Desse modo, concomitantemente com a revisão bibliográfica, realizou-se um estudo da documentação do PZ.

O estudo do ambiente PZ foi feito em duas etapas:

- Reimplementação das classes de sistemas de coordenadas;
- Implementação de subestruturação.

Na reimplementação do sistema de coordenadas foram estudados aspectos básicos da programação orientada a objetos, incluindo um estudo superficial das classes matriciais e classes de malhas. O resultado desse estudo é apresentado anexo.

Já a implementação de subestruturação serviu como um estudo mais profundo do ambiente e das classes de maior interesse ao desenvolvimento do projeto.

Pode-se destacar os seguintes aspectos diretamente ligados ao projeto:

- estudo da divisão de instâncias em geométricas e computacionais;
- estudo da estrutura de nós geométricos e suas respectivas conectividades computacionais;
- estudo de elementos geométricos e computacionais;
- estudo de malhas geométricas e computacionais;
- introdução aos conceitos de nós internos e externos;
- estudo de redução estática de nós internos sobre nós externos e
- operação inversa, dada uma solução nos nós externos como repassar esta aos nós internos;

Estes assuntos são de extrema importância para o trabalho, uma vez que pretende-se fazer a análise de erro em subdomínios, obtidos através da criação de malhas de *patches* de elementos, sendo aqui estudado e implementado toda a manipulação de estruturas de blocos de conectividades e de soluções, o referenciamento de uma malha filha para uma malha pai, etc.

O estudo realizado para a implementação das classes de subestruturação está anexo.

A documentação do ambiente PZ, bem como alguns artigos relacionados ao seu desenvolvimento podem ser encontrados no seguinte endereço internet: `http://labmec.fec.unicamp.br/~pz`.

Capítulo 3

Método dos Elementos Finitos - MEF

Diversos problemas de engenharia podem ser modelados matematicamente através de equações diferenciais.

A busca da solução para equações diferenciais por métodos analíticos é, em alguns casos, extremamente trabalhosa e, em boa parte dos casos, algo inviável, sendo utilizados métodos numéricos para aproximar a solução.

Existem diversos métodos para aproximar a solução de equações diferenciais tais como:

- Método das Diferenças Finitas;
- Método dos Elementos Finitos;
- Método dos Elementos de Contorno, etc.

O ambiente PZ, desenvolvido pelo grupo de pesquisa do Orientador, tem implementadas classes que permitem a aproximação de problemas 1D, 2D e 3D através da utilização do MEF e, com base neste ambiente, descrito em artigos tais como [11] que é desenvolvido esse trabalho.

Um dos problemas do MEF que é abordado neste trabalho, é que a qualidade da aproximação depende da discretização utilizada e da ordem dos polinômios aproximadores utilizados, podendo os resultados gerados não representar, de forma satisfatória, a realidade.

Para melhorar os resultados pode-se refinar a malha ou aumentar a ordem polinomial das funções testes utilizadas, existindo diversas formas de implementação desta adaptabilidade, sendo comuns as tipo: r, h, p e combinações dessas [19].

A escolha do método de adaptação é importante, pois a elevação excessiva da ordem de interpolação das funções de forma pode gerar instabilidade numérica e, conseqüentemente, apresentar resultados insatisfatórios.

Este trabalho busca o desenvolvimento de métodos genéricos, onde a auto-adaptabilidade é governada por um parâmetro de erro estimado através da comparação das aproximações de duas malhas com graus de refinamento hp distintos.

Assim, para a introdução dos conceitos relativos ao método dos elementos finitos será utilizado um problema modelo: a equação de Laplace em três dimensões. Ressalta-se que o método não é aplicável a apenas este problema.

3.1 O Método dos Elementos Finitos

O MEF é uma metodologia para aproximação de problemas diferenciais de valor de contorno. Tal metodologia consiste nos seguintes passos:

- dado a formulação diferencial do problema obter a formulação fraca do problema;
- aplicação do método de Galerkin para o espaço de funções adotado;
- resolução do sistema algébrico;
- análise da solução obtida.

Para descrever esta metodologia é utilizado um problema modelo, a expressão de Laplace, sendo desenvolvidas todas as etapas consideradas acima.

3.1.1 Problema Modelo - Equação de Laplace

Consideremos um domínio Ω em $\Re^3$, submetido a duas condições de contorno (CDC) em suas faces:

- Γ_D: Condição de contorno de Dirichlet, onde o valor da variável de estado é fixada;
- Γ_N: Condição de contorno de Neumann, onde o valor da função fluxo no contorno é fixada;

Este problema consiste em encontrar uma função $u(x, y, z)$, $(x, y, z) \in \Omega$ tal que:

$$\begin{cases} -\Delta u = f \ \forall \ (x, y, z) \in \Omega \\ u = u^D \ \forall \ (x, y, z) \in \Gamma_D \\ \frac{\partial u}{\partial n} = g \ \forall \ (x, y, z) \in \Gamma_N \end{cases} \tag{3.1}$$

onde:

- $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$[1] : Laplaciano;

[1] Foi adotado um sistema de coordenadas cartesiano apenas para a representação da expressão. A adoção de outros sistemas de coordenadas apenas alteraria a representação da expressão.

O Ambiente PZ, onde os métodos estão sendo implementados contempla os sistemas de coordenadas Cartesiano, Cilíndrico e Esférico.

- $\frac{\partial u}{\partial n} = \frac{\partial u}{\partial x} n_x + \frac{\partial u}{\partial y} n_y + \frac{\partial u}{\partial z} n_z$[2] : indica a normal ao contorno de Ω;
- $f = f(x, y, z)$, $(x, y, z) \in \Omega$: função definida para todo o domínio.
- $u^D = u_o(x, y, z)$, $(x, y, z) \in \Gamma_D$: condição de contorno Dirichlet, onde u_0 é o valor fixado para a variável de estado nesta região do contorno;
- $g = g(x, y, z)$, $(x, y, z) \in \Gamma_N$.: condição de contorno Neumann, onde $g(x, y, z)$ é o fluxo imposto naquela região do contorno;
- $\Omega = \Gamma_D \bigcup \Gamma_N$, $\Gamma_D \bigcap \Gamma_N = \emptyset$.

3.1.2 Formulação Fraca

Por simplicidade, inicialmente, será aqui mostrada a metodologia para a obtenção da formulação fraca para o caso da CDC Dirichlet homogênea, ou seja: $u_0(x, y, z) = 0 \, \forall \, (x, y, z) \in \Gamma_D$.

Desta forma, sendo $v = v(x, y, z)$ uma função teste utilizada, esta se anulará no contorno Γ_D, ou seja:

$$v(x, y, z) = 0 \; \forall \; (x, y, z) \in \Gamma_D \tag{3.2}$$

Multiplicando-se a expressão (5.1) pela função teste v e integrando o resultado sobre todo o domínio Ω, temos:

$$\int_\Omega -\Delta u \, v = \int_\Omega f \, v \tag{3.3}$$

Considerando que a função teste v é contínua, pode-se integrar por partes o lado esquerdo da expressão acima, tendo como resultado:

$$\int_\Omega \nabla u \nabla v - \int_{\Gamma_N} \frac{\partial u}{\partial n} v = \int_\Omega f \, v \tag{3.4}$$

onde ∇ denota o gradiente da função.

Observe que a integral no contorno ficou reduzida apenas ao trecho onde é aplicada a CDC Neumann pela simplificação adotada de que a função v é nula na região do contorno onde está aplicada a CDC Dirichlet (homogênea).

Desta forma, o problema inicial passa a ser representado agora pelo seguinte problema equivalente:

$$\begin{cases} \text{Encontrar } u(x, y, z) \in H^1(\Omega), \text{ tal que} \\ \int_\Omega \nabla u \, \nabla v = \int_{\Gamma_N} g \, v + \int_\Omega f \, v \\ \text{para toda função teste } v, \text{ tal que } v = 0 \text{ em } \Gamma_D \end{cases} \tag{3.5}$$

[2] n_x, n_y e n_z indicam os vetores unitários normais às superfícies de contorno.

Para que ocorra a equivalência entre a formulação fraca e a formulação forte há a necessidade de algumas definições extras a saber:

1. A função teste adotada deve pertencer ao *espaço de funções teste admissíveis*, sendo este espaço definido como:

$$V = \left\{ v(x,y,z) | v(x,y,z) = 0 \, \forall \, (x,y,z) \in \Gamma_D, v \in H^1(\Omega) \right\} \tag{3.6}$$

2. As funções f e g devem ser quadrado integrável (o quadrado da função deve ser integrável) em Ω e Γ_N respectivamente.

Outro aspecto importante a ser destacado é a possibilidade de representar a formulação fraca através de um termo bilinear e um termo linear conforme mostrado abaixo:

$$\begin{cases} b(u,v) = \int_\Omega \nabla u \, \nabla v \\ l(v) = \int_{\Gamma_N} g \, v + \int_\Omega f \, v \\ \text{e assim:} \\ b(u,v) = l(v) \end{cases} \tag{3.7}$$

Essa representação mostrar-se-á útil quando do estudo de convergência de estimadores de erro, onde a esta função será acrescido um termo de resíduo.

CDC Dirichlet Não Homogênea

Neste caso teremos que $u_0 \neq 0$ e assumindo que a função u_0 admita a "translação" $\widetilde{u}_0$, definida em todo o domínio Ω e satisfazendo todas as condições de regularidade impostas para a solução, temos que $\widetilde{u}_0$ deverá estar contida dentro do espaço de Sobolev $H^1(\Omega)$.

Desta forma, com a aplicação de $\widetilde{u}_0$ à formulação fraca obtida para o caso homogêneo teríamos:

$$\begin{cases} \text{Encontrar } u(x,y,z) \in \widetilde{u}_0 + V, \text{ tal que} \\ b(u,v) = l(v) \\ \text{para toda função teste } v \in V \end{cases} \tag{3.8}$$

onde:

$$\widetilde{u}_0 + V = \{ \widetilde{u}_0 + v, \, v \in V \} \tag{3.9}$$

Desta forma, tendo-se uma função $\widetilde{u}_0$ que satisfaça as condições impostas acima, temos que a função resultante u independe desta extensão, o que torna possível a seguinte substituição de variáveis: $u = \widetilde{u}_0 + w$, com $w \in V$ e satisfazendo todas as condições de continuidade impostas a u. Com isto, a formulação variacional pode ser reescrita da seguinte forma:

$$\begin{cases} \text{Encontrar } w(x,y,z) \in V, \text{ tal que} \\ b(w,v) = l(v) - b(\widetilde{u}_0, v) \\ \text{para toda função teste } v \in V \end{cases} \tag{3.10}$$

Ressalta-se que, desta forma, pode-se calcular a função $\widetilde{u}_0$ que satisfaz a condição de extensão e então, definindo-se : $l_{mod} = l(v) - b(\widetilde{u}_0, v)$, como sendo a parte linear modificada, podemos substituir na expressão acima e voltar a ter o mesmo padrão de expressão que aquele encontrado para as CDC homogêneas.

3.1.3 Método de Galerkin

Considerando que já temos a formulação fraca para o problema, o método de Galerkin consiste na restrição do espaço de funções teste V, utilizando um espaço $V_h \subset V$, espaço este composto por uma base de dimensão finita. Assim o problema a ser resolvido passa a ser:

$$\begin{cases} \text{Encontrar } u_h(x,y,z) \in \widetilde{u}_0 + V_h, \text{ tal que} \\ b(u_h, v_h) = l(v_h) \\ \text{para toda função teste } v_h \in V_h \end{cases} \tag{3.11}$$

Com relação à base de funções utilizada, esta pode ser representada por:

$$V_h = \{e_{hi}\},\ i = 1, 2, \ldots, N_h \tag{3.12}$$

onde $N_h = dim(V_h)$ indica a dimensão do espaço de aproximação.

Dentro deste espaço, procura-se a solução para o problema sob a forma da seguinte combinação linear:

$$u_h = \sum_{i=1}^{N_h} \alpha_{hi} e_{hi} \tag{3.13}$$

Os coeficientes α_{hi}, a serem determinados, são denominados *graus de liberdade* (d.o.f).

Substituindo a expressão acima na formulação fraca do problema e, ainda, adotando como funções teste a mesma base de funções utilizada para representar u_h, ou seja $v = e_{hj}$ $j = 1, 2, \ldots, N_h$, chegaremos à seguinte representação do problema:

$$\begin{cases} \text{Encontrar } \alpha_{hi}\, i = 1, 2, \ldots, N_h, \text{ tal que} \\ b(\sum_{i=1}^{N_h} \alpha_{hi}(e_{hi}, e_{hj})) = l(e_{hj}) \\ j = 1, 2, \ldots N_h \end{cases} \tag{3.14}$$

O método de Galerkin consiste na utilização do espaço de funções de interpolação e, através de manipulação da formulação variacional, transformá-la numa expressão algébrica.

Em princípio, a aproximação depende somente do subespaço adotado V_h, independendo da base de funções escolhida e_{hi}. Na prática, a escolha da base de funções afeta o condicionamento do sistema final, implicando diretamente sobre os erros de arredondamento, os quais podem ser significativos [6, 7].

Para finalizar, para o problema em questão, pode-se realizar algumas mudanças de notação conforme segue:

- Definindo-se a matriz de rigidez global S_{ij} como sendo:

$$S_{ij} = b(e_i, e_j) = \int_\Omega \nabla e_i \nabla e_j \tag{3.15}$$

- O vetor de carga modificado L_j^{mod} será dado por:

$$L_j^{mod} = l(e_j) - b(\widetilde{u}_0, e_j) = \int_\Omega f e_j + \int_{\Gamma_N} g e_j - \int_\Omega \nabla \widetilde{u}_0 \nabla e_j \tag{3.16}$$

- O vetor de carga original L_j é dado por:

$$L_j = l(e_j) = \int_\Omega f e_j + \int_{\Gamma_N} g e_j \tag{3.17}$$

E assim o problema pode ser escrito da forma usual:

$$\begin{cases} \text{Encontrar } \alpha_{hi}\, i = 1, 2, \ldots, N_h, \text{ tal que} \\ \sum_{i=1}^{N_h} \alpha_{hi} S_{ij} = L_j^{mod} \\ j = 1, 2, \ldots N_h \end{cases} \tag{3.18}$$

Este sistema pode ser representado por $[S] * [\alpha] = [L]$, ou seja, um sistema linear com inúmeras técnicas de resolução.

O Método dos Elementos Finitos - MEF

O MEF é um caso especial do método de Galerkin, onde uma sistemática para construção da base de funções a ser utilizada no método de Galerkin é definida. No caso deste trabalho específico, esta sistemática é utilizada para modificação da base de funções.

Essa construção leva em consideração a partição do domínio Ω em subdomínios, formando uma malha, sendo cada subdomínio denominado elemento finito e, para cada um destes elementos são introduzidas funções de forma Φ_k. Estas funções de forma tem como característica principal o fato de seu valor nos nós da malha assumir um valor binário, sendo 1 no nó para o qual a função foi definida e um conjunto de zeros para os outros nós da malha.

3.2 Definições

Antes de prosseguir com a discussão do método dos elementos finitos e entrar no campo da análise da qualidade da aproximação, faz-se necessário introduzir algumas notações que serão utilizadas na seqüência.

3.2.1 Precisão da Aproximação

O MEF fornece a função que mais se aproxima da função procurada[3], dentro do subespaço V_h escolhido. Entretanto, a escolha do subespaço mais adequado depende de variáveis poucas vezes conhecidas na prática.

Caso o subespaço escolhido V_h não seja adequado, a função encontrada não apresentará uma precisão satisfatória, necessitando então esses resultados de uma análise para verificar essa precisão.

Para todo subespaço escolhido, a restrição do espaço de funções teste implica na inserção de um erro. O erro da aproximação é dado por:

$$e(x) = u(x) - u_h(x) \tag{3.19}$$

onde:

$e(x)$: erro da aproximação no ponto x[4] ;

$u(x)$: valor da função u(real) calculada no ponto x;

$u_h(x)$: valor da função u_h(aproximada) calculada no ponto x.

3.2.2 Aspectos Relacionados ao Subespaço de Aproximação V_h

A escolha do subespaço V_h em programas que se utilizam do MEF é determinada pela malha de elementos finitos não sendo alterada durante o processo de cálculo.

Isto pode ser verificado pelo fato deste subespaço ser determinado pela discretização utilizada, representada pela malha de elementos finitos, sendo considerados tanto o "tamanho" do elemento, como também a localização de seus nós (pontos onde o problema será discretizado).

O outro parâmetro que influencia a escolha do subespaço é a base de função a ser utilizada, sendo esta pré definida para cada tipo de elemento que o programa contempla. Um tipó comum de base utilizada é a base polinomial.

Como já mencionado anteriormente, a precisão da aproximação está ligada diretamente ao subespaço V_h adotado, assim quando se quer melhorar esta aproximação existe a necessidade de se adaptar o subespaço de aproximação.

Em termos de definição, os parâmetros que influenciam a aproximação são:

[3] Quando o erro é medido pela norma de energia.

[4] x deve ser entendido com um ponto, com número de coordenadas compatível com a dimensão do problema e com o sistema de coordenadas adotado.

- h: parâmetro relacionado à discretização da malha, normalmente associado ao "tamanho" do elemento;
- r: parâmetro relacionado à disposição dos nós dos elementos na malha;
- p: parâmetro relacionado à ordem dos polinômios, de cada elemento, utilizados como base para o espaço V_h.

Dimensão e forma de um elemento

Tanto dimensão como forma do elemento são parâmetros que influenciam a aproximação.

Entretanto, para definir estes parâmetros, devemos ter em mente que diversos tipos de elementos finitos podem ser utilizados, sendo necessário assim uma padronização da nomenclatura utilizada. Aqui será adotada a definição dada em [1].

Assim, define-se:

- $h_k = max_l\{h_l\}$, $h_l = sup_{x,y \in k}|x - y|$, sendo h_l a máxima distância entre os nós do elemento k, distância esta tomada dois a dois. Desta forma este parâmetro indica o diâmetro do círculo circunscrito ao elemento;
- ρ_k: indica o diâmetro de um círculo inscrito ao elemento k;
- $\kappa_k = \frac{h_k}{\rho_k}$: índice que indica a forma do elemento, definido como a regularidade do elemento k.

3.2.3 Adaptabilidade e refinamento

Como mencionado acima, no caso da aproximação encontrada não ser aceitável, deve-se enriquecer o espaço de aproximação. O resultado do enriquecimento do espaço de aproximação é denominado espaço, ou malha, adaptado(a).

Assim, a adaptabilidade consiste no enriquecimento do espaço de funções V_h, onde este enriquecimento poder ser feito através do refinamento dos parâmetros h, r, p, ou ainda a combinação de destes.

O refinamento h consiste na redução do tamanho dos elementos componentes da malha, enquanto o refinamento p consiste na elevação da ordem dos polinômios da base de funções teste. O refinamento hp, objeto deste trabalho, consiste no refinamento destes dois parâmetros para a mesma malha.

Não será tratado do refinamento r neste trabalho. O refinamento r consiste no posicionamento ótimo dos nós que definem a malha de elementos finitos.

Capítulo 4

Auto - adaptabilidade e Estimação de Erros

Os estudos aqui realizados tem uma única motivação: desenvolver métodos que tornem automática a aproximação de soluções para problemas de valor de contorno, com precisão adequada, mesmo em caso onde as funções à aproximar não são conhecidas.

Para tal, a ferramenta utilizada é a adaptabilidade de espaços de aproximação e de funções para o método dos elementos finitos, tendo como parâmetro de análise um erro estimado através da diferença entre duas aproximações, sendo uma proveniente de uma malha com um refinamento *hp* qualquer e outra desta mesma malha refinada uniformemente em *hp*.

4.1 Adaptabilidade

A melhoria do espaço de aproximações, quando se tem o conhecimento prévio do padrão da solução, é relativamente fácil. Entretanto, quando esse padrão de solução não é conhecido, esta escolha torna-se difícil.

A busca de formas de melhoria do espaço de aproximações de maneira automática está intrinsicamente ligada à obtenção de um parâmetro que indique os parâmetros de refinamento que aumentem a qualidade da solução. O parâmetro que indica a qualidade da aproximação é o erro.

Entretanto, o erro real só pode ser obtido quando se tem o conhecimento prévio da solução do problema o que não é o caso prático, sendo assim necessário uma estimativa de erro.

A obtenção desta estimativa de erro é, de certo modo, o cerne de qualquer método auto-adaptativo, sendo a descrição de todos os aspectos envolvidos em sua obtenção descritos na seqüência.

4.2 Estimadores de Erro

Para a análise de uma aproximação gerada por meio de uma metodologia de elementos finitos há a necessidade de se ter um parâmetro para verificar sua qualidade, sendo o erro o parâmetro que melhor expressa essa qualidade.

Caso se conhecesse o resultado analítico de um problema, o cálculo do erro seria simples, bastando escolher o tipo de norma. Como a obtenção da solução analítica é algo que se tem apenas para os problemas utilizados na validação de códigos, sendo na prática, algo inviável para a maioria dos problemas nos quais são utilizados métodos numéricos, os métodos auto-adaptativos utilizam-se de uma abordagem na qual o erro da aproximação é estimado.

A teoria de elementos finitos mostra que o erro tende a zero à medida que a dimensão do espaço de interpolação é enriquecido (refinado) de maneira adequada, ver [21], ou seja a solução aproximada tende à solução real, quanto maior for o refinamento adotado.

Diversos parâmetros podem ser utilizados na estimação do erro, tais como comparação entre aproximações para diferentes espaços, comparação entre o gradiente calculado e um gradiente estimado, análise de derivadas, dentre outros. Entretanto, cuidado especial deve ser tomado na escolha do parâmetro para estimar o erro em cada caso, pois caso a curva de erro estimado não aproxime de maneira adequada a curva de erro real, pode-se ter resultados falsos, uma vez que a convergência do estimador de erros para zero não necessariamente indica que o erro real também está tendendo a zero (ver [1]).

Assim, a escolha de um estimador de erros deve ser precedida de algumas análises, conforme descrito na seqüência, de modo a garantir que o resultado final realmente aproxime a solução.

4.3 Aspectos Teóricos dos Estimadores de Erro

Para se estimar o erro parte-se do princípio de que quanto maior o enriquecimento de um espaço de aproximações, melhor será o resultado.

Assim, caso se compare o resultado obtido da utilização de um espaço de interpolação com parâmetros h_0 e p_0, com aqueles obtidos através de um espaço enriquecido $\tilde{h}_n$-$\tilde{p}_n$, onde $\tilde{h}_n \leq h_0$ e $\tilde{p}_n \geq p_0$, poderíamos calcular o erro entre as duas aproximações sabendo que a última é de melhor qualidade que a primeira.

Os estimadores de erro podem ser baseados em dois grupos: *a priori* e *a posteriori* [1].

Os estimadores de erro do tipo *a priori* representam um desafio para pesquisadores que se utilizam de análise numérica, sendo baseados em estimativas, como o valor da segunda derivada da função procurada por exemplo, sendo para tal necessário o conhecimento das funções envolvidas no problema bem como de seu comportamento. Entretanto, no caso de aplicações de maior complexidade, onde as funções analisadas não são conhecidas, esse tipo de estimador não se mostra interessante.

Os estimadores de erro do tipo *a posteriori* tem seu desenvolvimento relativamente recente, tendo como primeiro trabalho de destaque [2], onde o estimador de erro baseia-se nas aproximações da norma de energia do erro em cada elemento K da malha.

O uso da formulação de energia complementar para a obtenção *a posteriori* do erro foi feita por [5]. Entretanto, uma vez que seu método era baseado no computo global da solução, este tornava-se muito oneroso em termos de tempo de processamento. Este problema foi contornado por Ladevèze e Leguillon [18] que realizaram os cálculos baseados na energia complementar de cada elemento, além do uso do conceito de dados de equilíbrio de contorno.

Diversos trabalhos foram feitos utilizando diversos tipos de estimadores de erro, destacando-se o de Zienkiewicz e Zhu [26], cujo estimador de erro, baseado na obtenção da diferença entre o gradiente de uma solução suavizada e o gradiente calculado originalmente, mostrou-se eficaz, tornando-o muito popular.

4.4 Aspectos Matemáticos dos Estimadores de Erro

A qualidade de um estimador de erro, para uma solução de elementos finitos, está ligada à malha, considerando como malha não apenas à partição dos elementos, mas também o espaço de interpolação adotado.

Recorrendo às notações definidas anteriormente, temos;

$$e(x) = u(x) - u_h(x) \tag{4.1}$$

onde:

$u(x)$: solução exata no ponto (x);

$u_h(x)$: solução numérica (aproximada) no ponto (x).

Entretanto, não nos interessa apenas o erro em determinados pontos e sim no domínio analisado. Para tal é inserido o conceito da função norma.

A função norma é uma função que realiza o mapeamento de uma função para um número real. As principais normas utilizadas no estudo de estimadores de erro são:

- norma de energia $\|e\|^2 = a(u - u_x, u - u_x)$, com a: forma bilinear;
- norma L_2 : $\|e\|^2 = \int_\Omega |u - u_h|^2 \, dx$;
- norma L_∞ : $\|e\|_\infty = sup\{|u(x) - u_h(x)|, (x) \in \Omega\}$.

4.4.1 Propriedades dos Estimadores de Erros

Para analisar as características básicas de estimadores de erro do tipo *a posteriori*, recorre-se a formulação fraca do problema modelo, mostrado anteriormente na equação (3.5).

Utilizando o subespaço V_h tem-se:

$$B(u_h, v_h) = L(v_h) \, \forall \, v_h \in V_h \tag{4.2}$$

Desta forma, conforme [1], a função erro, apresentada em (4.1), pertence ao espaço V e satisfaz:

$$B(e, v) = B(u, v) - B(u_h, v) = L(v) - B(u_h, v) \, \forall \, v \in V \tag{4.3}$$

Ainda, dada a condição de ortogonalidade do erro para projeções de Galerkin (ver [1]), tem-se que:

$$B(e, v_h) = 0 \, \forall \, v_h \in V_h \tag{4.4}$$

Levando-se em consideração as expressões (4.3) e (4.4), integrando-se a primeira por partes, em cada elemento:

$$\int_k (\nabla e \nabla v) dx = \int_k (rv) dx + \oint_{\partial k} (v \nabla e_{|k} \overrightarrow{n_k}) ds \tag{4.5}$$

sendo:

e: função erro procurada;

v: função teste ou peso;

r: função residual ou resíduo $r = f + \triangle u$;

$\overrightarrow{n_k}$: vetor normal à face do elemento;

k: elemento em análise.

Resolvendo esta equação, considerando condições de continuidade das funções e de suas integrais, teremos:

$$\|e\| \leq C_1 \|r\|_{L_2(k)} + C_2 \|R\|_{L_2(\partial k)} \tag{4.6}$$

onde:

$R \approx \overrightarrow{n_k} \nabla e$,

C_1, C_2: constantes que dependem da malha, principalmente do tamanho do elemento (h_k);

Caso a aproximação do fluxo no contorno $(g + R)$, represente uma boa aproximação ao fluxo real, temos que a expressão acima fornece o limite superior para o erro em função do resíduo.

Ainda, definindo-se:

- η_k: como a estimativa de erro em cada elemento k.
- η: como a estimativa de erro na malha de elementos finitos.

A estimativa de erro na malha pode ser obtido através da seguinte expressão:

$$\eta = \sqrt{\sum_{k \in P} \eta_k^2} \tag{4.7}$$

onde P é a partição adotada e k um determinado elemento da partição.

Para que um estimador de erros seja utilizável este deve respeitar a seguinte propriedade:

$$C_1 \|e\| \leq \eta \leq C_2 \|e\| \tag{4.8}$$

ou seja, a estimativa do erro deve convergir para zero à mesma taxa que o erro real converge.

A qualidade de um estimador de erros é medida através do índice de efetividade, dado pela relação:

$$\Theta = \frac{\eta}{\|e\|} \tag{4.9}$$

Dada a importância do estimador de erros de Zienkiewicz e Zhu [26] este é descrito nos anexos do trabalho.

Capítulo 5

Método Base - Modelo Demkowicz e Devloo

O modelo auto-adaptativo que serve de base para o modelo que será implementado neste trabalho foi inicialmente proposto por Demkowicz, Rachowicz e Devloo em [8].

O método utiliza-se de um estimador de erros baseado na diferença das aproximações entre duas malhas, sendo com esta estimativa de erro definidos os elementos com erro significativo e que necessitam de refinamento.

A análise do padrão *hp* de refinamento é feita através da análise de cada aresta do elemento, conforme metodologia criada pelo Prof. Demkowicz (ver [8]).

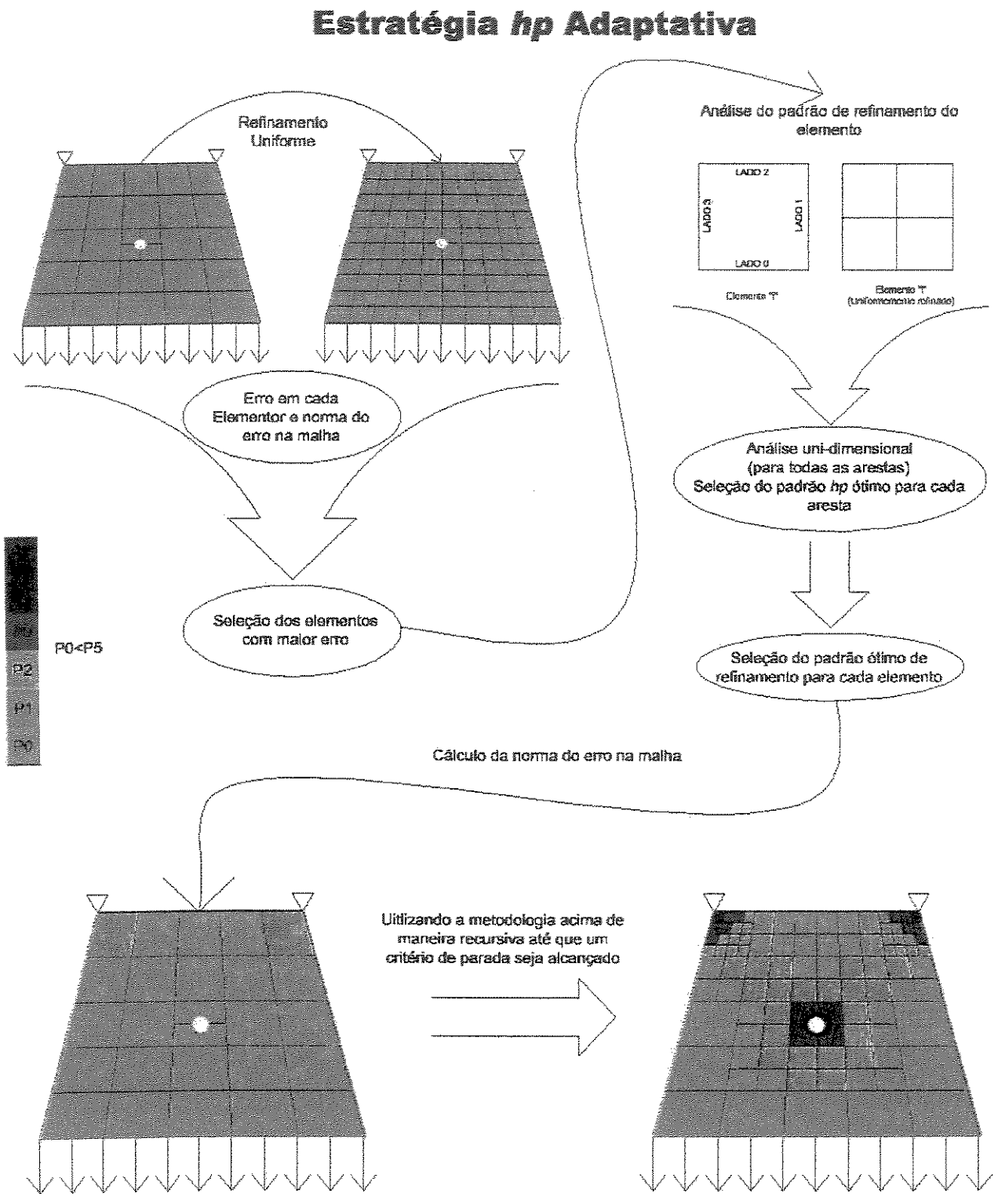
Esse modelo foi validado e está sendo incorporado em ambientes auto-adaptativos, tais como o 2Dhp90 e 3Dhp90 do TICAM - The University of Texas at Austin (ver [6, 7]). No ambiente PZ, esta técnica foi implementada pelo Prof. Devloo, com utilização de análise *mutigrid*.

Esquematicamente o método é mostrado na Figura (5.1), onde dada uma malha uniformemente refinada *hp* e a malha não refinada, é feita a estimativa do erro em cada elemento. Os elementos com erro "relevante" são selecionados para a análise do padrão de refinamento *hp*. Tendo-se o padrão ótimo de refinamento de cada elemento, este padrão é imposto na malha não refinada, obtendo-se desta forma a malha adaptada.

Sendo este processo realizado de maneira recursiva, até que algum critério de parada relacionado ao erro seja satisfeito, chega-se a malha adaptada com padrão *hp* requerido para obter a solução com a precisão estipulada.

Com relação aos resultados, o método apresentou taxas de convergência exponencial, conforme a teoria de elementos finitos preconiza, justificando assim a sua utilização neste trabalho.

Figura 5.1: Método Auto-adaptativo Base



5.1 Aspectos Teóricos

Existem trabalhos publicados com diversos tipos de estimador de erro, conforme pode ser observado em [20], [26] e [3], dentre outros.

O estimador de erro implementado para este método consiste no cálculo da diferença da solução de duas malhas com espaços de interpolação hierárquicos e distintos.

O valor estimado para o erro é calculado elemento a elemento, como sendo a diferença entre o valor do gradiente da solução na malha fina e o valor do gradiente da solução na malha grossa.

O valor do gradiente é facilmente obtido para cada elemento, sendo de fato necessário para os cálculos da aproximação.

5.1.1 Estimador para diferença entre malhas

No caso implementado, onde o cálculo do erro é feito através da diferença dos resultados entre duas malhas hierárquicas, prova-se a convergência do estimador de erro para o caso unidimensional. Esta demonstração para malhas unidimensionais é feita em [8] e reproduzido abaixo. Dado o seguinte problema modelo:

$$\begin{cases} u \in \widehat{u}_D + V \\ b(u,v) = l(v) \, \forall \, v \in V \end{cases} \tag{5.1}$$

onde:

$V \subset H^1(0,l)$: espaço de funções teste;

$\widehat{u}_D \in H^1(0,l)$: condição de contorno de Dirichlet;

$b(u,v)$, $l(v)$: formas bilinear e linear dos termos da equação diferencial do problema de valor de contorno, podendo ser escritas da seguinte forma:

$$\begin{cases} b(u,v) = \int_0^l (a.u'v' + bu'v + cuv)dx + \beta u(l)v(l) \\ l(v) = \int_0^l (fv)dx + gv(l) \end{cases} \tag{5.2}$$

com $a(x)$, $b(x)$, $c(x)$, $f(x)$ e β satisfazendo as condições de regularidade necessárias.

Dada uma malha $h-p$ com o seu respectivo espaço $V_{hp} \subset V$ a solução aproximada pelo método de Galerkin será:

$$\begin{cases} u_{hp} \in \widehat{u}_D + V_{hp} \\ b(u_{hp}, v_{hp}) = l(v_{hp}) \, \forall \, v_{hp} \in V_{hp} \end{cases} \tag{5.3}$$

Temos, para este caso, que o estimador de erros converge, sendo limitado por:

$$\|u - u_{hp}\|_{L_\infty} \leq C. \inf_{v_{hp} \in V} \|u - (\widehat{u}_D + v_{hp})\|_{L_\infty} \tag{5.4}$$

com $C > 0$, sendo uma constante que depende do espaço de interpolação (ver [8, pág. 4]).

5.1.2 Determinação do Padrão de Refinamento dos Elementos

Análise Uni-dimensional

A técnica implementada, apresentada em [8], consiste na minimização da projeção do erro de interpolação, devido ao refinamento *hp* em cada aresta do elemento.

O método tem como primeiro passo o cálculo do interpolante $u^n_{h,p}$, onde n denota a ordem do interpolante, sendo este obtido através da diferença entre uma solução proveniente da malha uniformemente refinada, $u_{\frac{h}{2},p+1}$, e a solução da malha original, $u_{h,p}$.

Tendo o interpolante calculado, o passo seguinte é a projeção da diferença entre a solução da malha refinada e do interpolante no espaço das funções teste das arestas dos elementos da malha, $V_h(e)$, e dada que a solução nos nós externos é igual para a malha refinada e a não refinada $u_{\frac{h}{2},p+1} - u^n_{h,p} = 0$ para estes nós.

Desta forma, o problema passa a ser encontrar a combinação *hp* que minimiza a diferença entre o interpolante e a projeção da solução obtida do refinamento *hp* no interior da malha.

Observe que por trabalhar apenas com espaços de funções e com projeções de espaços, esta metodologia pode ser aplicada a uma série de elementos, independente de sua topologia.

O objetivo do método é definir um refinamento, com parâmetros *hp*, com número de graus de liberdade menor que o número de graus de liberdade da solução uniformemente refinada *hp*, com resultados próximos a esta, ou seja minimizando esse erro.

As combinações nas quais será procurada a combinação *hp* ótimas são todas as combinações possíveis de p^1_{Kref} e p^2_{Kref}, sendo estas as ordens dos subelementos da aresta analisada, tais que:

$$p^1_{Kref} + p^2_{Kref} = p_K + 1 \tag{5.5}$$

Também é analisado o erro obtido devido ao refinamento p, sendo:

$$p_{Kref} = p_K + 1 \tag{5.6}$$

O erro entre a aproximação fornecida u e a aproximação calculada $\widetilde{u}$ no elemento K, utilizando o refinamento com número de graus de liberdade menor será:

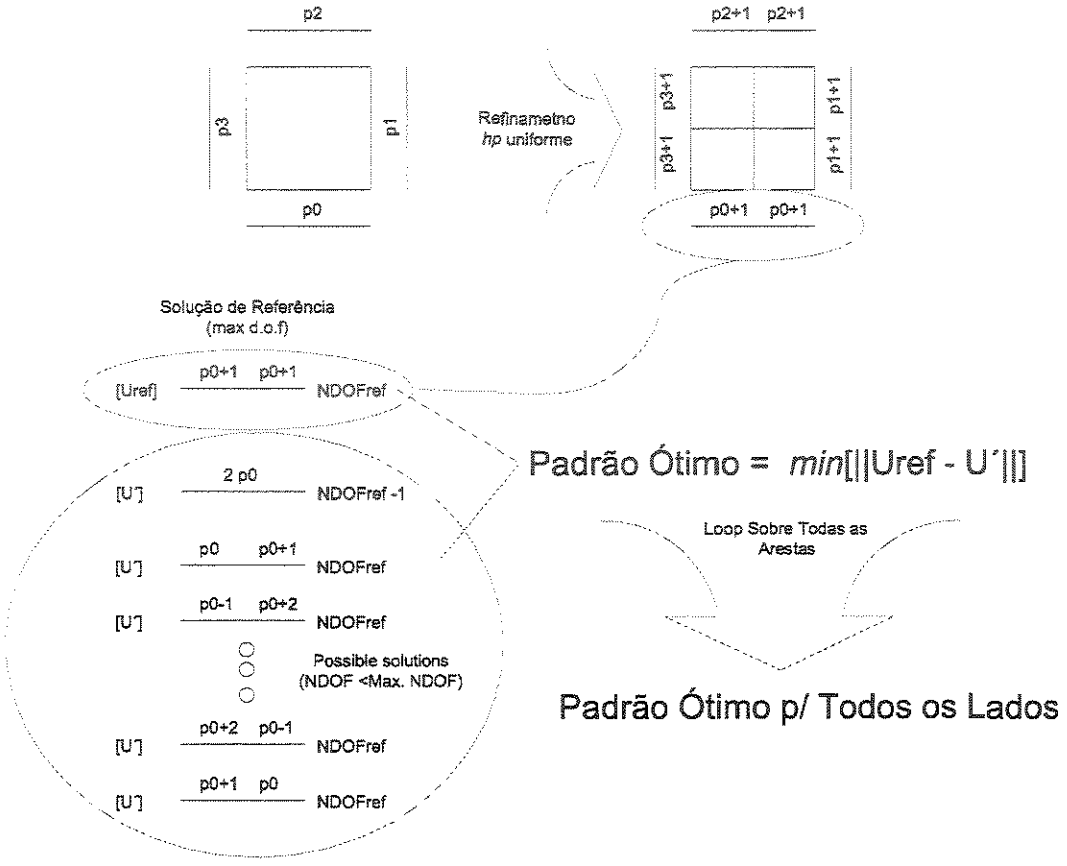
$$\|e\|^2_{L_2(K)} = \int_{-\Delta_x}^{\Delta_x} (u'_K - \widetilde{u_K}')^2 dx \tag{5.7}$$

A minimização desse erro, que representa um "resíduo" pode ser aproximada por:

$$\left\{ \begin{array}{c} \widetilde{u} \in H^1_0(-\Delta_x, \Delta_x) \mid R = \int_\Omega \psi'.(\widetilde{u}' - u').d\Omega = 0 \forall \psi \in H^1_0 \\ \widetilde{u}(-\Delta_x) = 0 \\ \widetilde{u}(\Delta_x) = 0 \end{array} \right\} \Rightarrow \int_\Omega \psi'_j.(\psi'_i.\widetilde{u})d\Omega - \psi'_j.u' = 0 \tag{5.8}$$

Algebricamente:

Figura 5.2: Definição do Padrão Ótimo de Refinamento para um Elemento



$$K = \int_{d\Omega_K} \psi_i' . \psi_j' dx \tag{5.9}$$

$$F = \int_{-\Delta_x}^{\Delta_x} \psi_i' . u' dx \tag{5.10}$$

$$\|e\|^2_{L_2(K)} = \int_{-\Delta_x}^{\Delta_x} \sum_i (u_i - \tilde{u}_i) . \psi_i' . \sum_j (u_j - \tilde{u}_j) . \psi_j' dx \tag{5.11}$$

A combinação de refinamento hp que apresentar o menor erro para cada aresta será retornada em termos dos parâmetros de refinamento hp para o subelemento 1 e hp para o subelemento 2, ou ainda um refinamento exclusivamente p.

Esquematicamente, o processo é demonstrado na Figura (5.2).

Dois pontos são destacados em [6] a respeito deste modelo de definição do padrão de refinamento:

1. Esta projeção do erro interpolado leva a padrões de convergência h ótimos e a padrões p próximos ao ótimo. Assim, sua minimização em todos os elementos deve conduzir a uma malha próxima à malha ótima;

2. A estimativa de erro não necessariamente necessita ser calculada globalmente, podendo isto ser feito localmente, elemento a elemento. É neste aspecto que este trabalho se apóia.

O aspecto da possibilidade de cálculo do erro e dos padrões de refinamento localmente é interessantes sob o aspecto da possibilidade de paralelização do método.

Determinação do padrão de refinamento para o elemento

A técnica descrita anteriormente mostra como obter os padrões de refinamento ótimos para cada aresta, sendo necessário compatibilizar estes resultados dentro de cada elemento e de seus vizinhos.

A análise do elemento é feita em cada nó, sendo necessárias as seguintes considerações:

- Caso qualquer aresta que contribua para o nó requeira refinamento h em seu padrão ótimo, o refinamento h é adotado, passando-se então à análise do padrão de refinamento p em cada subelemento. Caso não exista refinamento h nos padrões de refinamento ótimos é então adotado o refinamento p uniforme, com p igual ao maior refinamento p de todos as arestas;

- No caso de refinamento hp definem-se os padrões p_K^1 e p_K^2, levando-se em consideração as combinações nas arestas cujo erro de aproximação é maior e levando em consideração o número máximo de graus de liberdade admissível para o elemento refinado (ver 5.5, pág. 38).

Tendo-se refinado os elementos, a malha obtida é considerada como a malha adaptada.

5.2 Implementação do Estimador de Erros e da Auto - Adaptabilidade

O estimador de erros proposto, consistindo da diferença entre os resultados de duas malhas, está implementado em conjunto com métodos multigrid, onde a diferença entre os resultados de uma malha grossa e uma malha uniformemente refinada (h e p) é utilizada como estimativa de erro para a definição dos parâmetros de refinamento.

No código são implementados os seguintes passos:

1. Calcular a matriz de transformação da solução da malha grossa para a malha refinada;

2. De posse da informação de refinamento entre as duas malhas é feito o cálculo do erro, elemento a elemento, obtendo o erro na malha como sendo o somatório do erro dos elementos.

O refinamento *hp*, conforme ilustrado na Figura (5.3), é feito baseado em uma análise unidimensional, aresta a aresta, em busca de qual combinação de ordens p e refinamentos h, em um determinado elemento, melhor aproxima uma solução obtida através de um refinamento uniforme *hp*, com um número de graus de liberdade menor que esta.

Um aspecto ressaltado é a necessidade de ter-se uma solução relativa à malha uniformemente refinada em *hp*. Apesar de parecer um contra-senso, quando inserido em uma metodologia *multigrid*, torna-se razoável, pois nesse método os refinamentos são feitos de modo a minimizar as baixas freqüências do erro, enquanto o "desrefinamento" serve para reduzir as altas freqüências do erro.

Na versão atual não há classe específica para o cálculo do erro ou para a adaptabilidade tendo estes sido implementados como métodos da Classe *TPZMGAnalysis*, conforme será posteriormente descrito.

5.2.1 Classe TPZMGAnalysis

Esta classe, derivada da classe *TPZAnalysis*, realiza os procedimentos necessários para o cálculo da solução, dado um objeto do tipo malha computacional (maiores detalhes sobre a estrutura do ambiente PZ ver [10]).

A estrutura da classe é baseada em um vetor de malhas computacionais, relativas aos ciclos multigrid. Dessa forma, são acrescidos métodos para o gerenciamento desse vetor e também a manipulação do referenciamento entre uma determinada malha computacional e a malha geométrica.

Destaca-se que durante o processo, existe uma única malha geométrica, sendo esta referenciada pelas malhas computacionais refinadas. A Figura (5.4) ilustra um exemplo desse referenciamento.

Esse gerenciamento das referências da malha geométrica também é feito dentro da classe *TPZMGAnalysis*.

Os métodos implementados nesta classe são descritos na seqüência.

void AppendMesh (TPZCompMesh *mesh)

Adiciona um ponteiro para uma malha computacional **mesh* ao vetor de ponteiros para malhas computacionais.

Esse método é utilizado para acrescentar uma nova malha ao ciclo multigrid.

Figura 5.3: Esquema de funcionamento do método auto-adaptável baseado no estimador de erros proposto

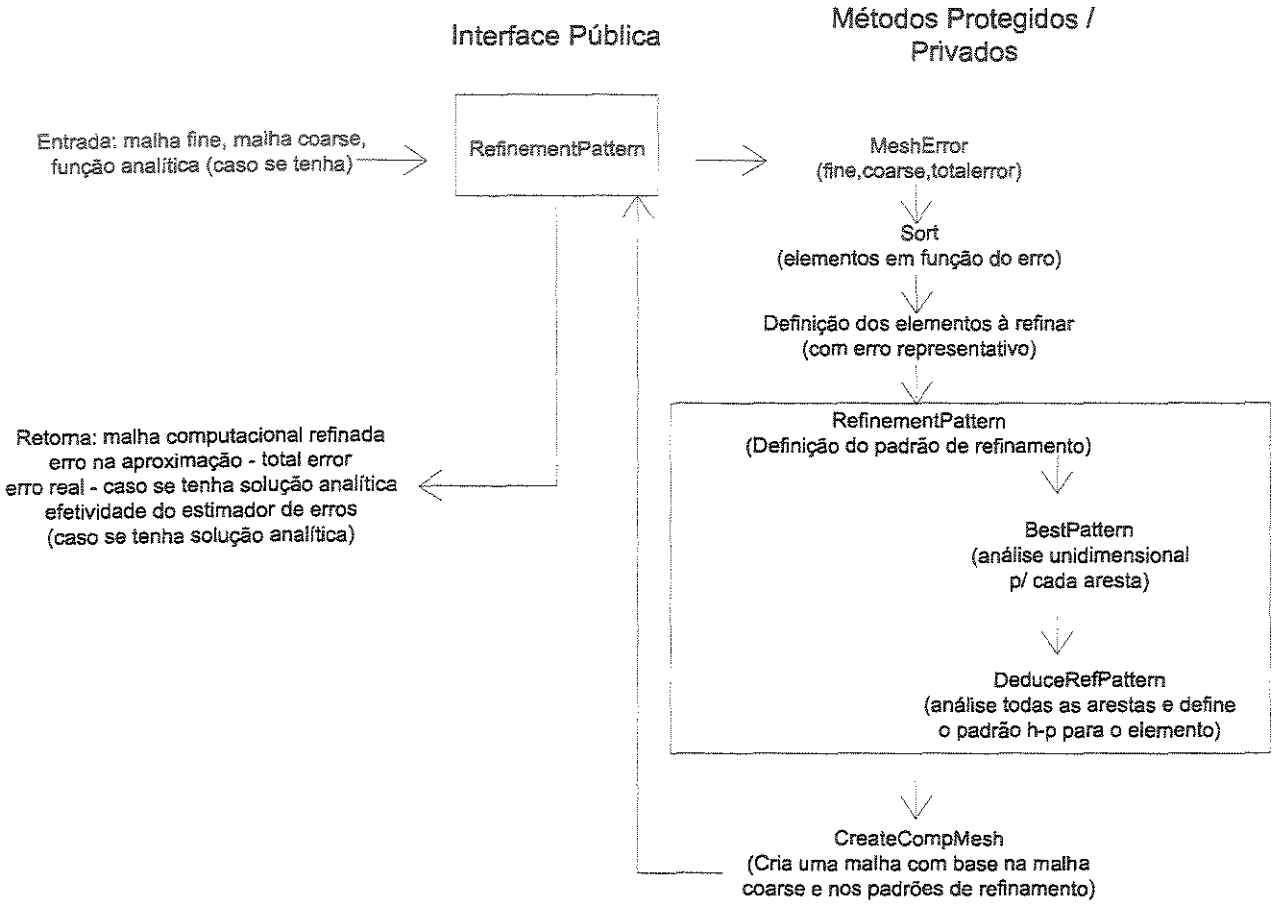
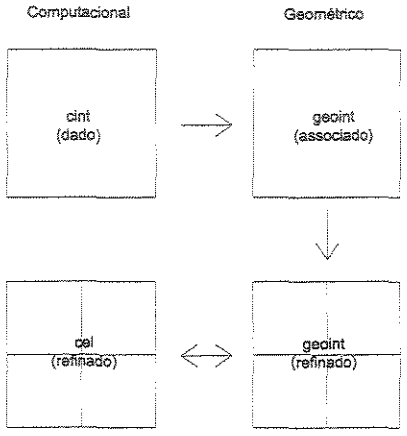


Figura 5.4: Referenciamento de malhas



TPZCompMesh* PopMesh()

Retira o último ponteiro para malha computacional do vetor de ponteiros para malhas computacionais, sendo também removidos o *solver* e o pré-condicionador associados.

TPZCompMesh* UniformlyRefineMesh (TPZCompMesh *mesh)

Realiza um refinamento uniforme *hp* em uma malha criada a partir de **mesh*, retornando a malha refinada, a qual será utilizada para o cálculo do erro e definição dos parâmetros *hp*, ótimos, os quais minimizam o erro com um número de graus de liberdade menor que o proveniente do refinamento uniforme.

No método multigrid este método pode ser utilizado para a obtenção da malha para uma primeira iteração, onde os parâmetros *hp* ainda não são conhecidos.

virtual void Solve ()

Utiliza o objeto *fSolve* para aplicar um procedimento de solução, que pode ser um método direto (Gauss, LU, LDLT etc) ou um método iterativo, preferível nos métodos multigrid, onde uma solução pode ser utilizada como pré-condicionador da próxima iteração.

TPZCompMesh* RefinementPattern (TPZCompMesh *fine, TPZCompMesh *coarse, REAL &totalerror, REAL &totaltruerror, TPZVec<REAL> &effect)

Retorna: uma malha refinada com parâmetros *hp* ótimos e com número de graus de liberdade menor que *fine*, o erro total da aproximação, caso exista uma solução de comparação (solução

exata) também é calculado o erro real da aproximação - *totaltruerror* e a efetividade do erro estimado *effect*.

Nesta função são realizadas as chamadas de maneira ordenada para: cálculo do erro entre as aproximações, verificação dos elementos à refinar, definição dos parâmetros de refinamento e criação da malha refinada. A seqüência de operações é descrita abaixo:

1. Calcular o erro entre as aproximações de *fine* e *coarse*: isto é feito através do método *MeshError (fine,coarse,ervec,fExact,truervec)*, sendo o erro nos elementos retornado no vetor de erros *ervec*;

2. Atribuir à solução de coarse o vetor *ervec*;

3. Ordenar os elementos em função do erro;

4. Calcular o erro total na malha;

5. Caso se tenha a solução exata, o valor do erro real pode ser calculado, tornando possível o cálculo do índice de efetividade do erro - *effect*;

6. Definição dos elementos que serão refinados: para o vetor de elementos, ordenado pelo erro, é feito o somatório da solução, que no caso contém o erro da aproximação, até que se atinja 65% do erro total, sendo os elementos computados até então aqueles que necessitam refinamento. O número 65% foi escolhido após alguns testes, com base na experiência do orientador e também do Prof. Leszek Demkowicz, autor da teoria de refinamento unidimensional.

7. Para os elementos selecionados acima, obtém-se seus elementos geométricos correspondentes, define-se a ordem de interpolação p desejada como sendo a ordem atual menos um. Cria-se um objeto do tipo *TPZOneDRef*, o qual analisa arestas em busca das ordens de refinamento ótimas, conforme será descrito posteriormente, e realiza-se a chamada para o método *AnalyseElement*, o qual, para cada elemento, analisará os requisitos de refinamento de suas arestas e verificará qual o refinamento ótimo a ser utilizado no elemento;

8. Com os parâmetros de refinamento específicos para cada elemento, estes são passados para método *CreateCompMesh*, o qual retornará a malha computacional refinada segundo os parâmetros *hp* passados.

void MeshError (TPZCompMesh *fine, TPZCompMesh *coarse, TPZVec<REAL> &ervec, void (*f)(TPZVec<REAL> &loc, TPZVec<REAL> &val, TPZFMatrix &deriv),TPZVec<REAL> &truervec)

Este método calcula a diferença entre duas aproximações e caso seja fornecida a solução exata, calcula também o erro real da aproximação.

Os parâmetros do método são:

- *fine*: malha refinada;
- *coarse*: malha com ordem de refinamento menor que fine;
- *ervec*: vetor contendo o erro entre as aproximações em cada elemento;
- *void (*f)(loc,val,deriv)*: função que pode conter a solução analítica para o problema, sendo passado em *loc*, o ponto, em *val* o valor da função e em *deriv* o valor de sua derivada;
- *truervec*: vetor contendo o erro real da aproximação. Só é calculado caso *(*f)* não seja nulo;

Para o cálculo do erro são feitos os seguintes procedimentos para cada elemento da malha *fine:*

1. Verifica-se se o elemento é proveniente de um espaço interpolado (*TPZInterpolatedElement*);
2. Procura-se na malha *coarse* o elemento que contenha o elemento interpolado identificado acima e o respectivo elemento geométrico associado a este;
3. A definição do nível de refinamento é feita através de procedimento recursivo de obtenção do elemento pai do elemento *fine*, até que este seja igual ao elemento *coarse* associado, fato que irá ocorrer pelo refinamento ser hierárquico. Tendo-se o elemento pai de *fine*, correspondente ao elemento *coarse,* é possível a identificação do elemento geométrico correspondente a este.
4. Com os elementos geométricos, identificados acima, é possível calcular a transformação entre estes elementos geométricos e seus respectivos elementos mestres - parâmetro dimensão.
5. Os valores de dimensão são passados como parâmetro para classe *TPZTransform*, a qual será descrita posteriormente, de tal forma que o objeto gerado possa calcular a transformação entre os elementos mestres associados aos elementos *fine* e *coarse.*
6. Com o objeto transformação calculado acima, é possível aplicar essa transformação aos pontos de integração do elemento da malha *fine*, de tal forma a levar esses ao mesmo espaço do elemento *coarse*, e assim tornar possível o cálculo da diferença da aproximação entre os dois elementos.
7. O cálculo do erro no elemento é feito através do método *ElementError*, descrito na seqüência.

REAL ElementError (TPZInterpolatedElement *fine, TPZInterpolatedElement *coarse, TPZTransform &tr, void (*f)(TPZVec<REAL> &loc, TPZVec<REAL> &val, TPZFMatrix &deriv),REAL &truerror)

Neste método é calculado o erro entre a aproximação de dois elementos interpolados, um relativo à malha *fine* e outro proveniente da malha *coarse*, com a transformação entre estes passada através do objeto *tr*. Caso tenha-se também a solução exata *(*f)* o erro real da aproximação é retornado em *truerror*.

O procedimento para o cálculo é o seguinte:

1. As soluções passadas em cada elemento são transformadas, como o realizado em pós processamento normal, nas funções de forma e respectivo gradiente relacionados ao elemento.

2. Com as funções calculadas, estas são rotacionadas de tal forma que seus eixos coincidam, sendo os valores da solução recalculados em relação a esses eixos. Para o cálculo da solução é utilizada uma regra de integração de ordem 20 (20 pontos de integração). Isso é feito pelo fato da precisão desse cálculo poder interferir no valor do erro.

3. Com os valores das soluções referenciando aos mesmos eixos, torna-se possível calcular o erro, o qual pode ser obtido através da norma L_2, ou através da semi-norma do erro (diferença entre os valores dos gradientes das soluções).

void AnalyseElement (TPZOneDRef &f, TPZInterpolatedElement *cint, TPZStack<TPZGeoEl *> &subels, TPZStack<int> &porders):

Neste método são realizados os cálculos e chamadas necessárias à definição dos parâmetros de refinamento $h-p$. Para tal, são dados um objeto do tipo *TPZOneDRef* - f e um elemento computacional - *cint*.

O resultado da análise será retornado em *subels* - refinamento h e *porders* - refinamento p.

A análise consiste em:

- Obter os seguintes dados de *cint*: elemento geométrico associado, subelementos associados, número de nós, número de arestas, solução U nos nós etc;

- Para cada aresta:

  - obter a lista de sub elementos associados ao lado,
  - calcular a dimensão destes (será igual para todos, pois estes são provenientes de um refinamento uniforme)
  - obter os dados dos blocos associados aos nós para possibilitar a obtenção da solução U a estes associados,

  - realizar a análise unidimensional com base em f - *BestPattern*, passando como parâmetro a matriz com a solução nos blocos dos nós associados aos subelementos de *cint*, os identificadores dos nós destes subelementos, a ordem p atual destes subelementos e a dimensão dos subelementos - *delx*. Conforme será descrito posteriormente, serão retornados os parâmetros p_1 e p_2 ótimos ou a necessidade de refinamento p ($p_2 = -1$), sem refinamento h;
  - criar um objeto de padrão de refinamento - *TPZRefPattern* e inserir os resultados da análise unidimensional nele;
  - inserir o padrão de refinamento deste lado em um vetor, o qual conterá os padrões de refinamento de cada aresta do elemento;

- Com o vetor de padrões de refinamento para todas as arestas do elemento, este é passado como parâmetro para o método *DeduceRefPattern*, o qual analisará todos os parâmetros e definirá o padrão de refinamento do elemento.

void DeduceRefPattern (TPZVec<TPZRefPattern> &refpat, TPZVec<int> &cornerids, TPZVec<int> &porders, int originalp)

Esse método calcula as ordens de interpolação p para todos os subelementos, sendo aqui feita a análise dos dados provenientes do estudo unidimensional de cada aresta, transformando estas informações em padrões de refinamento para o elemento.

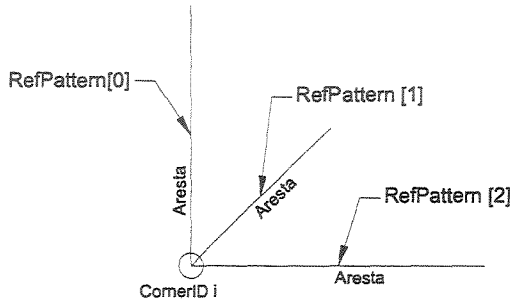
Os parâmetros de entrada são:

- *refpat*: vetor contendo os padrões de refinamento para cada aresta do elemento;
- *cornerids*: vetor contendo os identificadores dos nós, os quais estão referenciados dentro dos objetos de padrão de refinamento;
- *porders*: retornará o vetor contendo as ordens de refinamento p de cada aresta do elemento;
- *originalp*: ordem de refinamento p original do elemento, proveniente do refinamento unidimensional;

As operações aqui realizadas, com notação ilustrada na Figura (5.5) são as seguintes:

- Calcula-se o erro total no elemento, como sendo a soma dos erros obtidos através do refinamento ótimo de suas arestas;
- Os componentes do vetor de padrões de refinamento, com erros da ordem de 10^{-3} do erro total são desprezados;

Figura 5.5: Dedução do padrão de refinamento em um nó



- Verifica-se a necessidade de refinamento h em qualquer um dos pontos analisados. Caso isso ocorra em pelo menos um ponto, será realizado refinamento h. Caso não seja necessário o refinamento h, o refinamento p será uniforme, com p igual ao maior refinamento p constante no vetor de padrões de refinamento;

- No caso de necessidade de refinamento h, os nós de canto são ordenados em função do erro ótimo obtido quando da análise unidimensional;

- Em seguida, novamente são desprezados nós com erro pouco significativo em relação ao erro total do elemento;

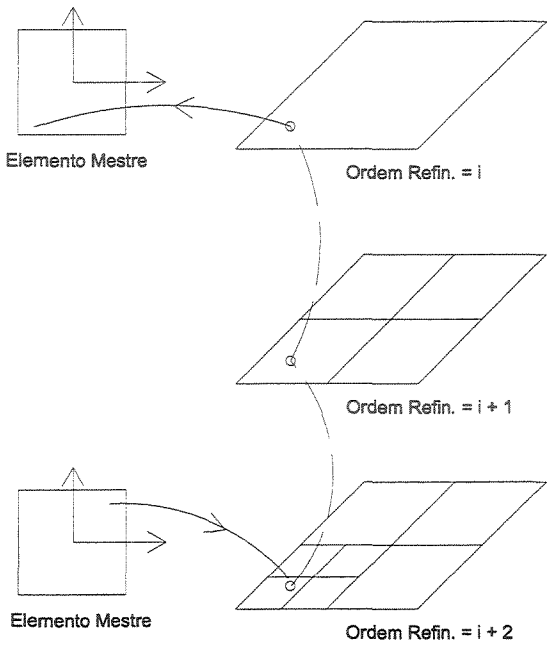
- Para cada nó, verifica-se se este corresponde a um dos nós do padrão de refinamento e, em caso afirmativo, verifica-se se a ordem h a ser adotada é a relativa ao subelemento 1 h_1 ou relativa ao subelemento 2 h_2;

- A ordem p é redefinida como sendo a ordem original - *originalp*, dividida por 2 e acrescida de um, de modo a compatibilizar o número de graus de liberdade anterior e novo.

void Sort (TPZVec<REAL> &vec, TPZVec<int> &perm) e void HeapSort (TPZVec<REAL> &sol, TPZVec<int> &perm)

Métodos de ordenação de componentes de vetores, utilizados para ordenar lados de elementos em termos de sua ordem de refinamento ou de seu erro;

Figura 5.6: Transformação linear de espaços



TPZCompMesh* CreateCompMesh (TPZCompMesh *mesh, TPZVec<TPZGeoEl *> &gelstack, TPZVec<int> &porders)

Retorna uma malha computacional criada através de uma dada malha computacional **mesh*, do vetor de elementos geométricos que devem ser refinados *gelstack* e de um vetor de ordem de refinamento *p porders*, relativo a cada elemento passado em *gelstack*.

Com os elementos de *gelstack* e com as ordens de refinamento, é possível a criação dos elementos computacionais através do método *CreateCompEl*. Os elementos criados são inseridos, com a devida ordem de refinamento, em uma malha computacional a qual referencia *mesh*.

5.2.2 Classe TPZTransform

Implementa a transformação linear de espaços R_n para R_n, com $n \leq 3$.

As operações feitas dentro da classe consistem na definição da operação de mapeamento, a qual consiste na aplicação de uma translação somada a uma rotação, conforme mostrado na Figura (5.6).

A rotação é definida através da matriz *fMult* enquanto a translação é definida pela matriz *fSum*. A matriz é armazenada em *fStore* tendo dimensões *fCol* e *fRow*.

5.2.3 Classe TPZTransfer

Implementa a transferência de soluções e resíduos entre espaços "hierárquicos", sendo acumuladas as transformações realizadas a cada refinamento da malha, possibilitando o cálculo da matriz de transformação.

5.2.4 Classe TPZMGSolver

Implementa a seqüência de operações do método multigrid, incluindo ciclos multigrid, armazenamento da matriz de transferência, vetor de malhas e malha grossa inicial.

5.2.5 Classe TPZOneDRef

Essa classe tem por objetivo definir o melhor refinamento *hp* em um elemento linear. Esse elemento linear, no caso do ambiente PZ, inclui arestas de elementos bi-dimensionais e tri-dimensionais.

O processo consiste em: dada a solução em elementos uniformemente refinados *hp*, de ordem $n+1$, verificar qual o refinamento do elemento pai, de ordem n, que melhor aproxima a solução uniformemente refinada.

Para tal, os parâmetros que devem ser fornecidos à classe são:

- Solução U nos elementos refinados, provenientes do elemento em análise, após este sofrer um refinamento uniforme *hp*;
- Ordem das funções de forma dos elementos pequenos p_1 e p_2;
- Dimensão do elemento refinado Δ_x, com a qual será possível calcular a transformação entre elementos refinados e não refinados;

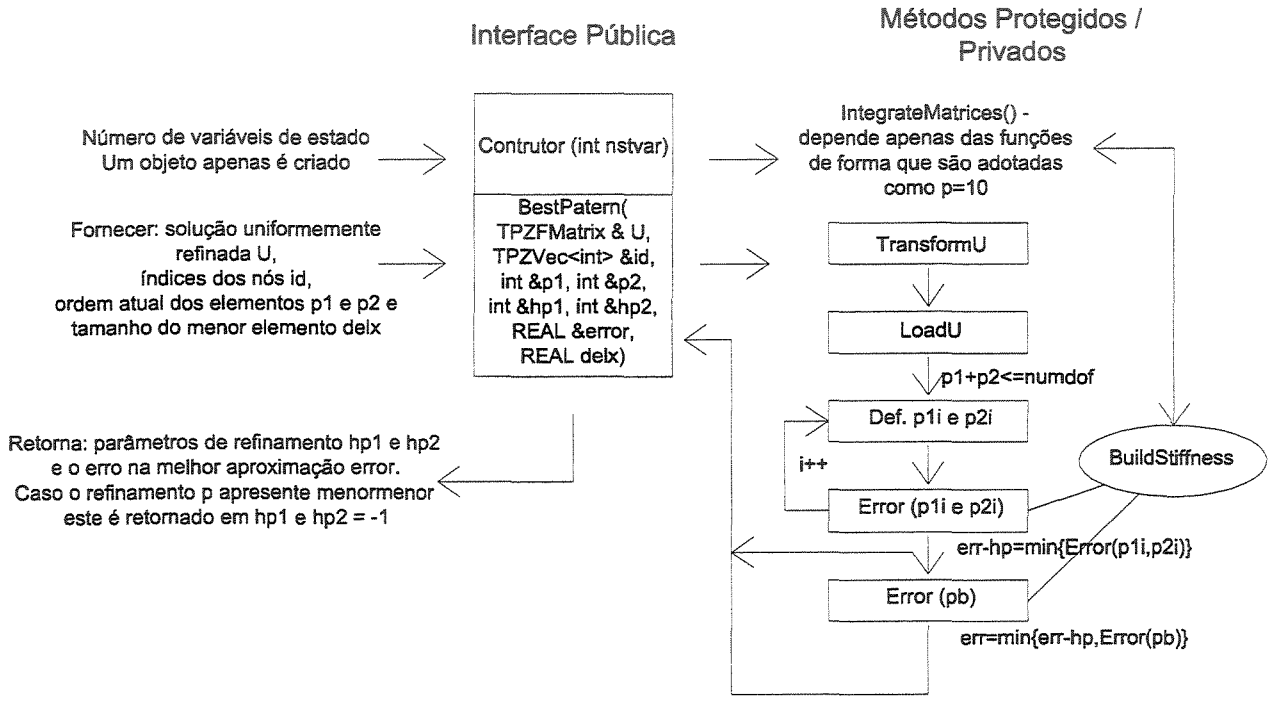
Com estas informações é possível determinar as funções de forma dos elementos, bem como as transformações entre elementos refinados e o elemento não refinado, sendo possível a transferência da solução fornecida nos subelementos para o elemento pai.

A Figura (5.7) ilustra a forma de utilização desta classe, sendo seus métodos descritos na seqüência.

Construtor: TPZOneDRef (int nstate)

No construtor são criadas todas as matrizes K (matriz de rigidez sem considerar os nós com restrições), F (vetor de carga), U (solução procurada) e M (matriz auxiliar), dos elementos refinados e dos elementos grandes, considerando um tamanho fixo, sendo definida como ordem máxima de interpolação 10. Esse valor foi escolhido com base na prática, uma vez que ordens de interpolação muito elevadas podem causar grandes oscilações entre os nós, afetando a estabilidade numérica do código.

Figura 5.7: Interface da Classe TPZOneDRef



A matriz M tem a função de armazenar os valores relativos à solução, incluindo os nós com restrição.

O parâmetro *nstate* indica o número de variáveis de estado que terá o vetor solução.

Feito o dimensionamento desses vetores, estes já são inicializados pelo método *IntegrateMatrices()*.

void IntegrateMatrices()

No ambiente PZ, os aspectos geométricos e computacionais de malhas e elementos são separados. Desta forma, dada a ordem de refinamento de cada elemento, que no caso é fixo, e também as funções de forma, que são calculadas com respeito ao elemento mestre em termos de bases hierárquicas, é possível a montagem das matrizes de rigidez dos elementos refinados e do elemento não refinado.

Assim, são feitos os seguintes cálculos:

$$fMS_1S_1 = \int_{\widetilde{\Omega}_1} \psi_{s_i}.\psi_{s_j}.d\widetilde{\Omega}_1 \tag{5.12}$$

$$fMS_1B = \int_{\widetilde{\Omega}_1} \psi_{s_i}.\psi_{b_j}.d\widetilde{\Omega}_1 \tag{5.13}$$

$$fMS_2B = \int_{\widetilde{\Omega}_2} \psi_{s_i}.\psi_{b_j}.d\widetilde{\Omega}_2 \tag{5.14}$$

$$fMBB = \int_{\widetilde{\Omega}_1+\widetilde{\Omega}_2} \psi_{s_i}.\psi_{s_j}.d\widetilde{\Omega} \tag{5.15}$$

$$fKS_1S_1 = \int_{\widetilde{\Omega}_1} \psi'_{s_i}.\psi'_{s_j}.d\widetilde{\Omega}_1 \tag{5.16}$$

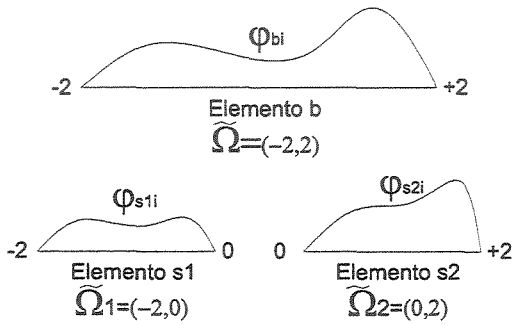
$$fKS_1B = \int_{\widetilde{\Omega}_1} \psi'_{s_i}.\psi'_{b_j}.d\widetilde{\Omega}_1 \tag{5.17}$$

$$fKS_2B = \int_{\widetilde{\Omega}_2} \psi'_{s_i}.\psi'_{b_j}.d\widetilde{\Omega}_2 \tag{5.18}$$

$$fKBB = \int_{\widetilde{\Omega}_1+\widetilde{\Omega}_2} \psi'_{b_i}.\psi'_{b_j}.d\widetilde{\Omega} \tag{5.19}$$

As funções ψ_{s_i} e ψ_{b_i}são as funções de forma dos elementos refinados e do elemento não refinado, respectivamente, sendo seu cálculo realizado através do método *TPZShapeLinear::Shape1D(int ptx, int order, TPZMatrix phi, TPZMatrix dphi, int id)*, onde *ptx* é o ponto onde se quer calcular a função, *order* indica a ordem com a qual devem ser criadas as funções de forma, *phi* é a matriz onde serão armazenados os valores da função *phi* e *dphi* será a matriz onde serão armazenados os valores calculados para a derivada de *phi*. A Figura (5.8) ilustra a notação utilizada.

Figura 5.8: Notação para as funções de forma



REAL BestPattern (TPZFMatrix &U, TPZVec<int> &id, int &p1, int &p2, int &hp1, int &hp2, REAL &hperror, REAL delx)

Este método é o único método público, além do construtor, da classe e faz a chamada de todas as funções necessárias à obtenção dos parâmetros $h-p$ ótimos.

Os parâmetros de entrada são:

- U: matriz com os resultados obtidos através do refinamento uniforme $h-p$;
- *id*: vetor com os identificadores dos nós, utilizado para a correção das funções de forma. Isso é necessário em função da implementação atual considerar que uma função inicia-se no nó de identificador menor e termina no nó de identificador maior;
- *p1* e *p2*: ordens dos polinômios dos subelementos 1 e 2, obtidos após o refinamento uniforme;
- *hp1* e *hp2*: ordens dos polinômios para os subelementos 1 e 2 os quais minimizam o erro e o número de graus de liberdade;
- *hperror*: retornará o erro correspondente aos parâmetros $h-p$ ótimos calculados;
- *delx*: dimensão geométrica dos subelementos. Esse valor é necessário para o cálculo das transformações entre subelementos e elemento não refinado (cálculo do Jacobiano da transformação linear).

Com estes dados e com os valores das matrizes calculadas quando da criação do objeto, serão realizadas as seguintes operações:

- Verificação das funções de forma dos elementos - método *TransformU;*
- Transporte da solução dos subelementos para o elemento não refinado - método *LoadU*;
- Nova verificação das funções de forma - *TransformU;*
- Cálculo do número de graus de liberdade *numdof*, sendo adotado o valor do elemento de maior número de graus de liberdade decrescido de um, pois buscamos uma solução com um número de graus de liberdade menor do que a solução uniformemente refinada;
- Define-se ordem $P_1 = 1$ e $P_2 = numdof + 1 - P_1$;
- Calcula-se o erro desta primeira aproximação - *besterror* ;
- Varia-se $p_1$de 2 até $numdof - 1$, com $p_2 = numdof + 1 - P_1$ e para cada combinação calcula-se o erro - método *Error(int p1, int p2)*. Caso o erro obtido em uma determinada iteração seja menor que *besterror*, esse é redefinido com o novo valor e $P_1 = p_1$ e $P_2 = p_2$. Ao final de todas as iterações ter-se-á a melhor combinação $h - p$ e o menor erro possível;
- A última verificação que é feita é a não utilização do refinamento h, e o cálculo do erro - método *Error(int pb)*. Caso o erro obtido seja menor, a ordem p_b calculada é definida como ótima e retornada em hp_1 sendo hp_2 definido como -1, de modo a identificar a não utilização do refinamento h;

Os métodos citados acima são descritos na seqüência.

TransformU(TPZFMatrix &U, TPZVec<int> &id, int p1, int p2)

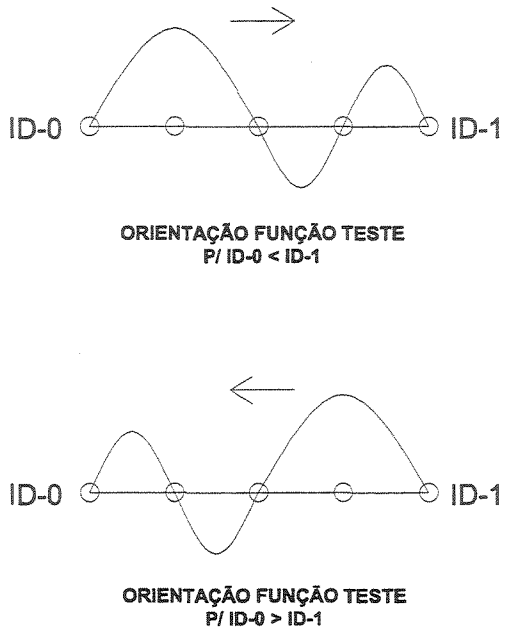
O método *TPZOneDRef::TransformU(TPZFMatrix &U, TPZVec<int> &id, int p1, int p2)* realiza uma compatibilização das funções de forma de ordem ímpar. Isso acontece pelo fato destas funções poderem satisfazer as condições necessárias para uma função de forma de duas maneiras distintas, ou sendo côncava ou convexa no trecho entre o nó inicial e o primeiro nó intermediário.

A convenção adotada no PZ é que a função sempre deve ser orientada do nó de menor identificador para o nó de maior identificador, conforme mostrado na Figura (5.9).

LoadU (TPZFMatrix &U, int p1, int p2, REAL delx)

Neste método, a solução U, proveniente do refinamento uniforme $h - p$, é passada para os elementos refinados da classe, considerando que os elementos refinados terão ordem de refinamento $p_b = p_1 + p_2 - 2$, onde p_1 e p_2 são as ordens das funções de forma dos elementos provenientes do refinamento uniforme. O parâmetro *delx* representa a menor dimensão dos

Figura 5.9: Convenção para funções de ordem ímpar



elementos refinados, sendo utilizada para o cálculo da transformação entre os elementos refinados e o elemento não refinado.

Com a transformação calculada, aplica-se o Jacobiano da transformação às matrizes de rigidez já calculadas e tem-se a matriz de rigidez dos elementos refinados em termos do espaço do elemento não refinado.

Através de um pós-processamento normal, dada a solução no elemento refinado, é possível encontrar-se a solução no elemento não refinado.

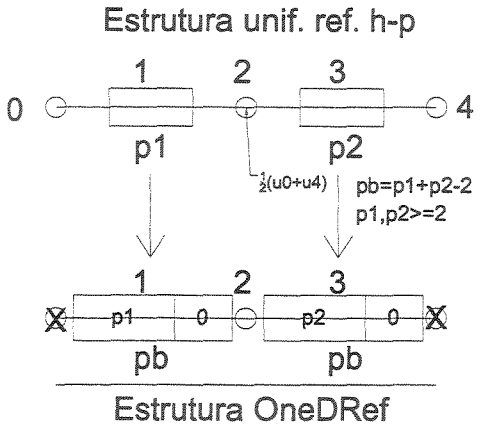
Outro procedimento feito é tornar nula a solução nos nós extremos (nó 0 e nó 2), de modo a possibilitar a solução de um problema tendo como malha os dois elementos refinados. Desta forma, considera-se que o erro nesses pontos será nulo, sendo considerado o erro relativo ao restante do domínio. Os procedimentos são ilustrados na Figura (5.10).

void BuildStiffness (int p1, int p2, TPZFMatrix &stiff)

Esse método calcula a matriz de rigidez, com base nas ordens de refinamento p_1 e p_2, retornando a matriz gerada em *stiff*.

Os valores da matriz são aqueles calculados no método *IntegrateMatrices()*, entretanto, por motivo de consideração de condições de contorno Dirichlet homogêneas, os nós de extremidade da malha, onde a resposta é fixada em zero, não são considerados quando da alocação da matriz *stiff*.

Figura 5.10: LoadU - transferência de blocos



REAL Error (int p1, int p2)

Este método retorna o erro mínimo que pode ser obtido utilizando-se ordens de refinamento p_1 e p_2 nos subelementos.

Para tal é realizada a seguinte seqüência de operações:

1. Gera-se uma matriz de rigidez *stiff* - método *BuildStiffness(p1,p2,stiff)*, tendo como parâmetros p_1 e p_2;

2. Gera-se o vetor de resíduos *rhs*, tendo o cuidado de utilizar a mesma estrutura de alocação de blocos utilizado na geração da matriz de rigidez;

3. Calcula-se a solução do problema $stiff * u_{ref} = rhs$. Para tal utiliza-se um *solver* do tipo LDLT, sendo o resultado retornado na matriz *rhs*;

4. É calculada a diferença entre a solução inicial e a solução aqui calculada $\Delta u = u - u_{ref}$;

5. O erro é então calculado através de (??), substituindo Δu e lembrando que : $K_{ij} = \sum_i \sum_j \psi_i \psi_j$, corresponde a matriz de rigidez global, temos:

$$erro = \sum_i \sum_j \Delta u.K.\Delta u$$

REAL Error(int pb)

Este método calcula o menor erro que pode ser obtido sem a utilização de refinamento h, com um refinamento geral p_b, sendo realizadas as seguintes operações:

1. Gera-se um vetor de resíduos rhs_b de ordem $p_b - 1$, com base no vetor de resíduos original (obtido do elemento uniformemente refinado $h - p$);

2. Gera-se uma matriz de rigidez para o elemento não refinado. Como já existe uma matriz de rigidez calculada para os elementos refinados, esta é aproveitada, sendo sob ela aplicada a transformação adequada (aplicação do Jacobiano)

$$stiff = \frac{1}{\Delta x} * fKSS$$

3. Calcula-se a projeção da solução do elemento refinado no elemento não refinado, sendo essa considerada sua solução para o trecho de sobreposição com o elemento refinado.

4. Resolve-se o sistema gerado $stiff * u_{res} = rhs_b$;

5. Ajustam-se os blocos de matrizes, de tal forma a se chegar ao padrão utilizado pelo ambiente PZ, onde os blocos dos nós com restrição não são considerados durante esse cálculo, pois seus resultados são conhecidos e nulos;

6. Calcula-se $\Delta u = u - u_{res}$;

7. Da mesma forma que descrito no método anterior, o erro é dado por:

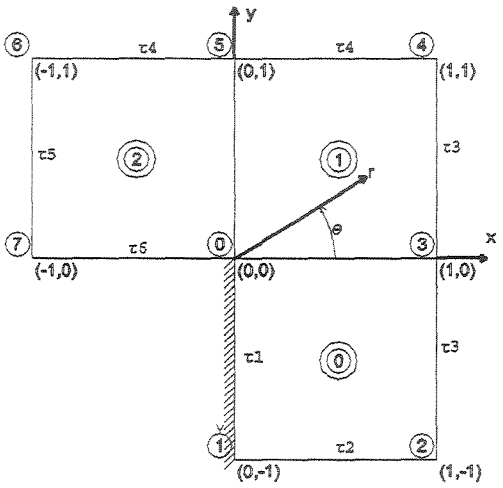
$$erro = \sum_i \sum_j \Delta u.K.\Delta u$$

struct TPZRefPattern { int fId[3]; int fp[2]; int fh[2]; REAL fhError; REAL fError; }

Esta é uma estrutura de armazenamento de informações de refinamento unidimensional, sendo considerado sempre que a estrutura consiste de um elemento refinado uniformemente em $h - p$, tendo assim dois subelementos gerados. Seus objetos armazenam as seguintes informações:

- *fId[3]*: armazena os identificadores dos nós iniciais e finais dos subelementos;
- *fp[2]*: armazena a ordem de refinamento p para cada um dos subelementos;
- *fh[2]*: armazena a ordem de refinamento h para cada um dos subelementos;
- *fhError*: armazena o menor erro obtido através de refinamento h podendo ter ou não refinamento p:
- *fError*: menor erro obtido entre *fhError* e o erro obtido por refinamento p simples:

Figura 5.11: Problema de Laplace



5.3 Resultados obtidos

Para a validação deste modelo foi utilizado o problema de Laplace, com dados indicados na Figura (5.11), onde as condições de contorno são colocadas como sendo o gradiente da solução. O problema é dado por:

$$\Delta u = 0 \; x \in \Omega \tag{5.20}$$

As condições de contorno aplicadas são tais que $\frac{\partial u}{\partial n}$ corresponde a solução exata:

$$u(r,\theta) = \sqrt[3]{r}\sin\left(\frac{1}{3}\left(\theta + \frac{\pi}{2}\right)\right) \tag{5.21}$$

Esta solução analítica possibilita o cálculo do erro real em cada elemento e desta forma verificar o índice de efetividade do estimador de erros implementado.

O método multigrid implementado no ambiente PZ apresentou para este modelo a taxa de convergência apresentada na Figura (5.12).

A efetividade obtida é mostrada na Figura (5.13), apresentando um configuração assintótica. No caso do modelo implementado esta efetividade chegou a aproximadamente 80% para 1500 equações. Dadas as restrições de equipamento (disponibilidade de memória), o problema modelo foi adaptado em 25 passos sendo as verificações feitas até 1600 equações.

O tempo de processamento acumulado (tempo contado a partir do início do programa), para cada passo do processo é apresentado na Figura (5.14), sendo o número de equações representado na abscissa igual ao número de equações da malha original a ser adaptada, ou

Figura 5.12: Convergência do Erro - Modelo de Estimação Global

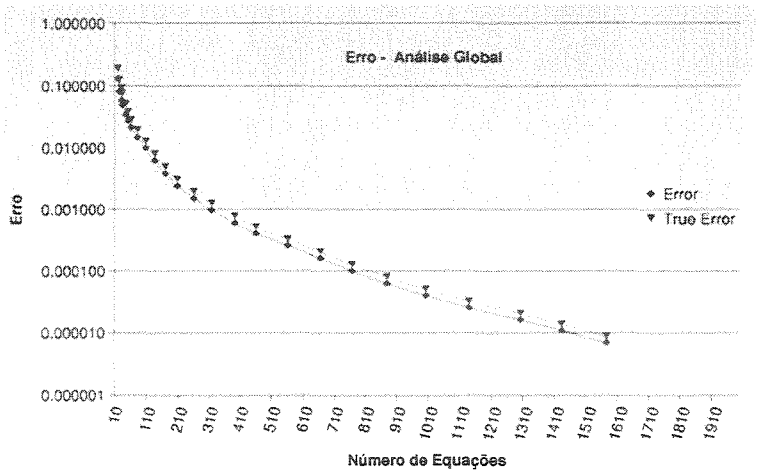


Figura 5.13: Efetividade do Estimador de Erros - Modelo de Estimação Global

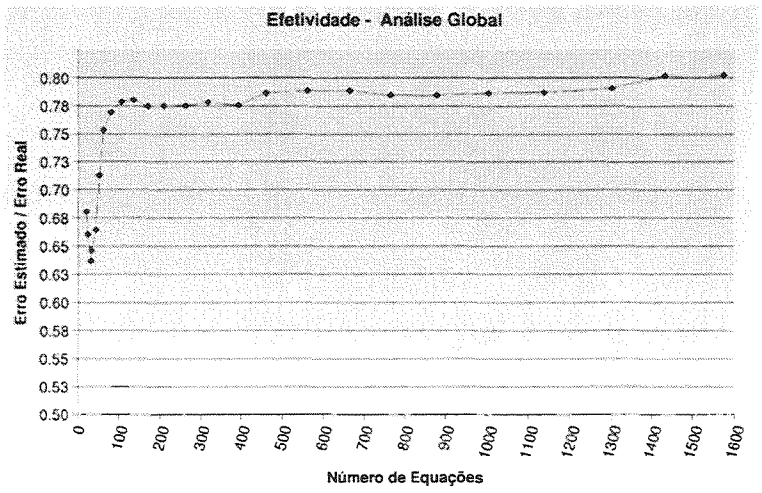
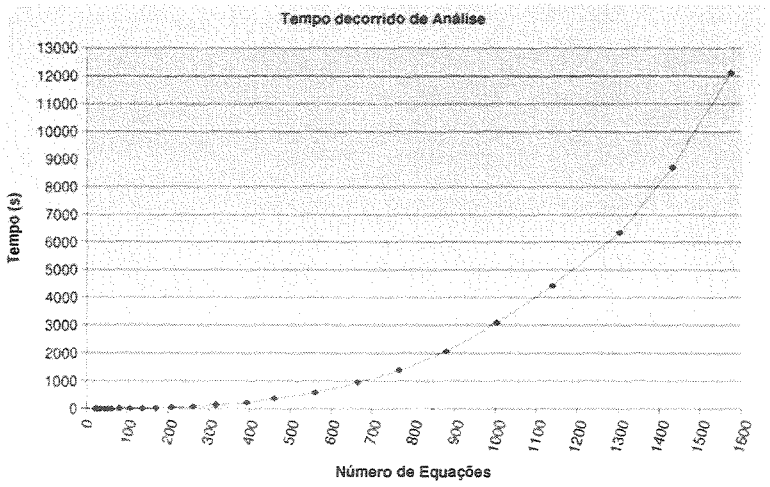


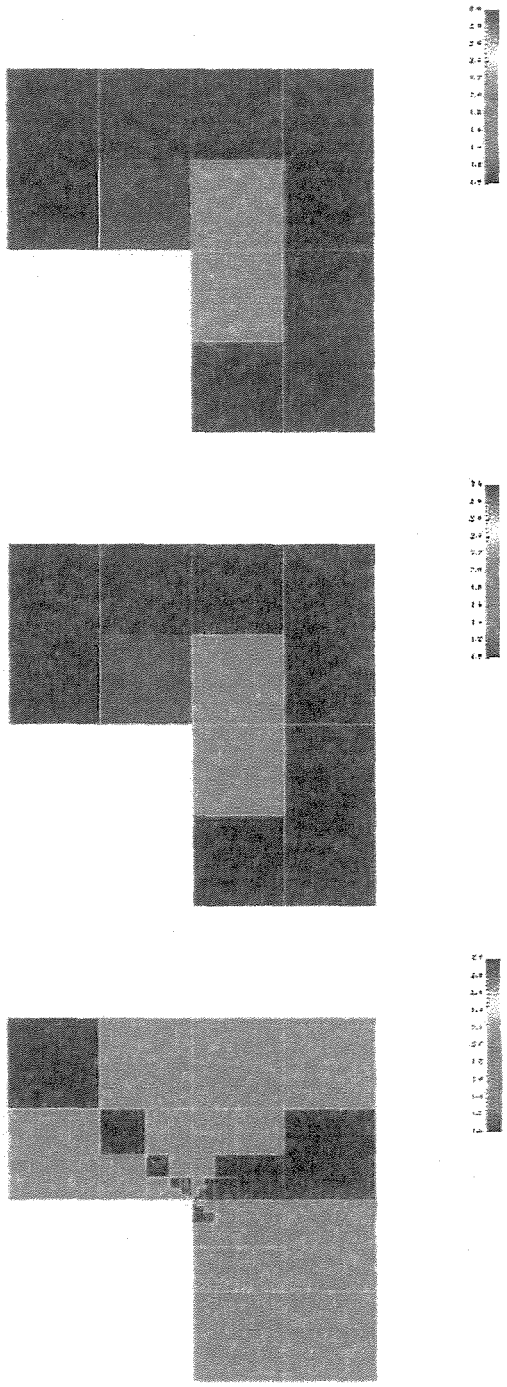
Figura 5.14: Tempo de Processamento - Modelo de Estimação Global



seja é representado o tempo necessário para a obtenção da malha adaptada para uma malha com aquele número de graus de liberdade. O equipamento utilizado neste teste foi um PC DUAL Athlon 1.2 GHz, com 1.0 GB de memória RAM padrão DDR (233MHz).

As malhas obtidas nos passos adaptativos são apresentadas na Figura (5.15), onde são apresentadas as malhas após 2, 5 e 10 passos adaptativos.

Figura 5.15: Malhas adaptadas após 2, 6 e 10 passos adaptativos - Escala de cores indica ordem de refinamento p



Capítulo 6

Estimador de Erro através de Sub-malhas

Conforme descrito anteriormente, a metodologia base para o trabalho é aquela implementada em [8] e já implementada no ambiente PZ.

A diferença entre a metodologia já implementada e a metodologia aqui proposta consiste na forma da obtenção da solução refinada para estimação de erro e obtenção do padrão de refinamento do elemento.

O método consiste na utilização de espaços reduzidos para o refinamento uniforme da malha e posterior aproximação do resultado local.

A obtenção do erro na aproximação e análise do padrão de refinamento é feita localmente, sendo para tal a malha dividida em um conjunto de elementos, a partir de agora denominados "elementos de referência do *patch*", os quais formam uma partição da malha, ou seja, o conjunto resultante da união destes elementos é a própria malha, enquanto o conjunto intersecção destes elementos é um conjunto vazio.

A partir destes elementos é formado um *patch* de elementos, os quais servirão de base para a criação de uma malha clone. Esta malha clone tem solução inicial igual à solução da malha original.

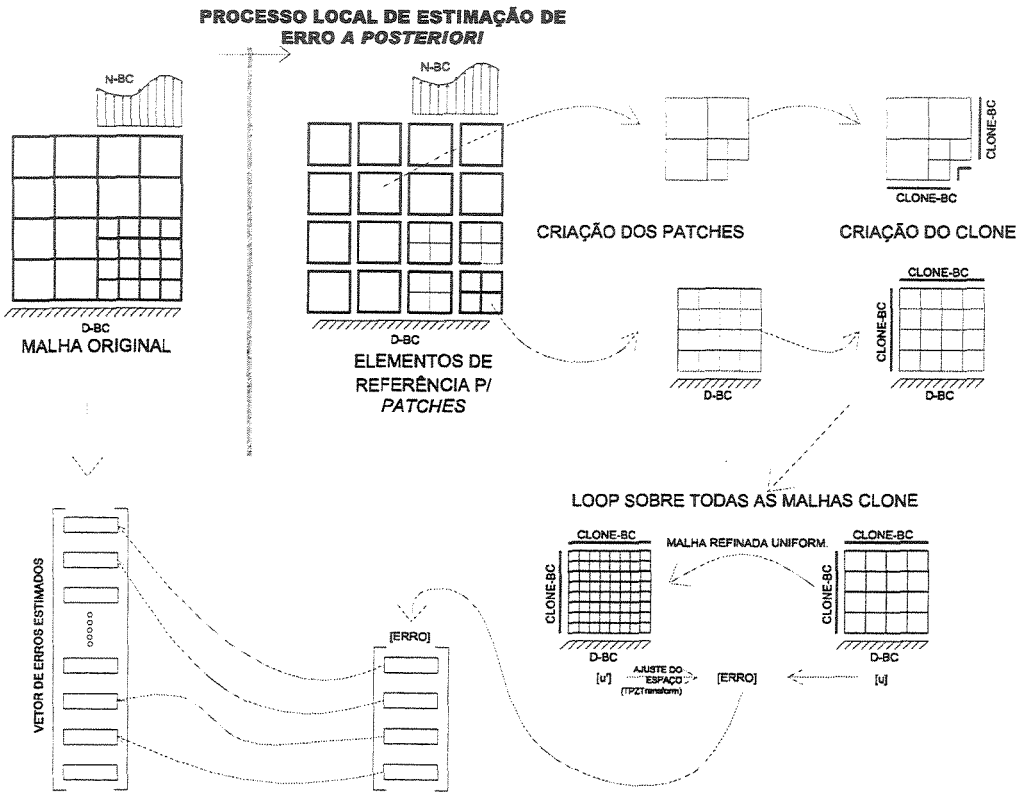
Da mesma forma que para a metodologia proposta anteriormente, a malha clone sofre um refinamento *hp* uniforme sendo sua solução utilizada como valor de referência para estimar o erro nos elementos associados ao elemento de referência da malha clone.

Este gerenciamento de elementos de referência para o *patch*, *patches* de elementos, malhas clone, malhas clone refinadas etc é, com certeza, a contribuição tecnológica deste trabalho sendo descrita em pormenores na seqüência. O funcionamento do método é mostrado de maneira esquemática na Figura (6.1).

Em primeiro lugar são abordados os aspectos relacionados ao estimador de erros utilizando um subespaço, ao invés do espaço de aproximações do problema.

Na seqüência são descritos os aspectos envolvidos na escolha dos elementos de referência para o *patch*, a formação dos *patches*, a criação das malhas clone e seu refinamento, a obtenção do erro e com base neste a escolha do padrão de refinamento dos elementos e a adaptação da malha original.

Figura 6.1: Estimador de Erros - Obtenção Local



Na seção posterior são enfatizados os aspectos relativos à implementação, descrevendo-se as classes implementadas bem como suas variáveis, métodos e funções.

Para finalizar são mostrados os exemplos utilizados na validação do código, os resultados obtidos e comentários a respeito destes.

6.1 Estimador de Erros Baseado em Subespaços

Quando se considera a utilização de subdomínios para obtenção de um estimador de erros, o primeiro fato que deve ser atentado é que não estará sendo levado em consideração o erro devido ao restante da malha, ou seja, estará se desprezando o erro por acoplamento. A parte desprezada do erro será maior quanto menor for o subespaço utilizado em relação ao espaço do domínio do problema.

De fato, no caso de utilização do menor subespaço possível, que seria a análise elemento a elemento, não há garantias teóricas para que o estimador venha a convergir para o erro real, ver [1].

Intuitivamente, quanto maior o subdomínio considerado, mais o resultado se aproximará do resultado apresentado na equação (4.8), onde a convergência é demonstrada.

Assim, um problema a ser estudado futuramente será a definição de qual dimensão de subespaço utilizar, de modo a compatibilizar a convergência do método com o custo computacional gerado pela resolução de um subespaço refinado.

6.1.1 Subespaço proposto - Definição do *patch* de elementos

Diversas metodologias podem ser adotadas na escolha, sendo aqui definido como critério básico a necessidade dos elementos de referência para a formação dos *patches* formarem uma partição da malha conforme já mencionado.

De maneira a facilitar o entendimento deste estudo, faz-se necessária a definição de alguns termos que serão utilizados de agora em diante:

- Elemento de Referência do *Patch*: consiste de um elemento geométrico da malha original, tendo como característica especial o fato de ser o maior elemento ancestral a ter vizinhos. O elemento de referência pode ou não ter sido refinado, tendo subelementos[1]. A forma de escolha dos elementos de referência será descrita adiante com maiores detalhes;

[1]Os elementos no ambiente PZ tem duas representações: uma geométrica e outra computacional. Os elementos geométricos guardam referências para seus subelementos e também para o seu elemento pai. Estas características tornam possível a identificação de toda a "genealogia" de um elemento necessária às operações descritas. Maiores detalhes podem ser encontrados em [10].

- *Patch*: é um conjunto de elementos formado pelo elemento de referência do *patch* ao centro, e por todos os elementos vizinhos ligados a ele diretamente, ou cujos nós sofram contribuição do elemento de referência ou de um de seus subelementos.

A forma de obtenção dos elementos de referência do *patch*, aqui adotada, é a seguinte:

Algorithm 1 Obtenção dos Elementos de Referência dos *Patches*

1. Definir uma lista vazia de Elementos de Referência do *Patch* - *Stack <Elementos> ElRefPatch*
2. Para todos os elementos da malha original (malha a ser adaptada)
   (a) Criar uma lista de elementos ancestrais ao elemento, inserindo o elemento analisado na lista
   (b) Inserir todos os ancestrais do elemento na lista de ancestrais, de tal forma que o elemento analisado seja o primeiro da lista e o elemento ancestral mais antigo, proveniente da malha original, seja o último
   (c) Percorrer a lista de elementos ancestrais do final para o início e caso o elemento analisado tenha vizinhos com nível de refinamento igual ao seu, este elemento será definido como elemento de referência do *patch*, para o elemento analisado
   (d) Verificar se o elemento definido anteriormente já está na lista de elementos de referência do patch, em caso afirmativo passar para o próximo elemento da malha original, em caso negativo, inserir o elemento na lista

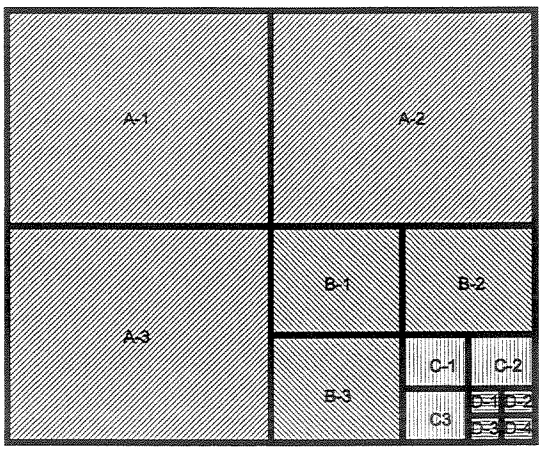
Tendo-se os elementos de referência para os *patches*, o passo seguinte é, para cada *patch*, identificar os elementos vizinhos ao seu elemento de referência, ou cujos nós contribuam para o elemento de referência do *patch* ou de algum de seus subelementos.

Como exemplo, analisando a malha mostrada na Figura (6.2), temos:

Nível	Denominação
0	A: $A_1, A_2, A_3, (A_4)$
1	$B : B_1, B_2, B_3, (B_4)$
2	$C : C_1, C_2, C_3, (C_4)$
3	$D : D_1, D_2, D_3, D_4$

Os elementos de nível "0" são os elementos da malha original, sendo um desses elementos A_4 refinado, dando origem aos elementos de nível "1" - elementos B_i, dentre os quais o elemento B_4 foi refinado , e assim sucessivamente. Caso fôssemos montar o "*patch*" de alguns elementos selecionados, teríamos:

Figura 6.2: Ordem de Refinamento



Elemento analisado	Elemento de referência do *patch*	Componentes do *patch*
A_1	A_1	Todos elementos
B_2	A_4	Todos elementos
C_3	B_4	Subelementos de A_4
D_4	D_4	Subelementos de C_4

6.2 Implementação do Estimador de Erros Local

Ressalta-se que o conhecimento do funcionamento do ambiente PZ é de extrema importância para o entendimento dos métodos aqui implementados, principalmente naqueles relativos a estrutura de malhas e elementos.

Como exemplo fundamental está o caso de malhas geométricas e computacionais.

No ambiente PZ uma malha geométrica referencia a apenas uma malha computacional, podendo, entretanto, várias malhas computacionais referenciarem a mesma malha geométrica.

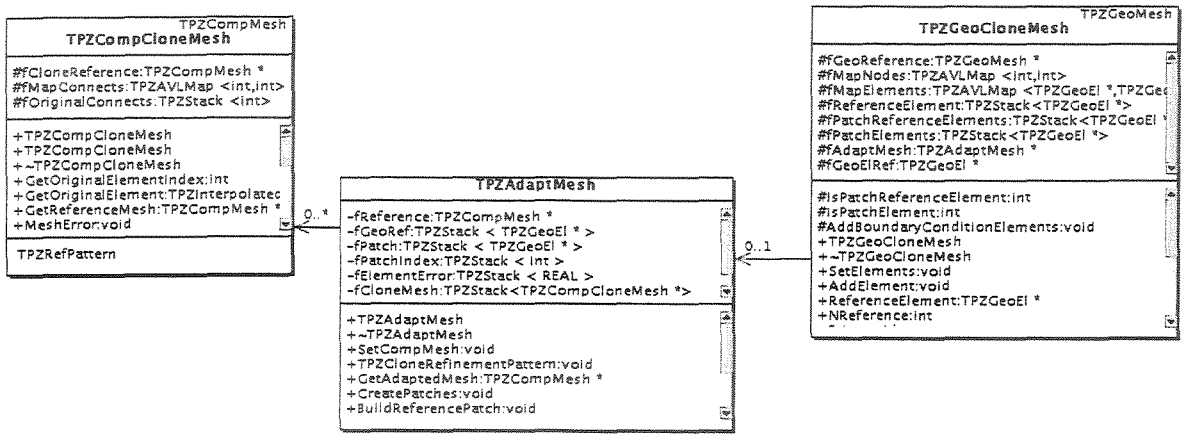
Este dado é fundamental na implementação é feita na seqüência, pois este gerenciamento de apontadores é utilizado constantemente para a obtenção de informações corretas, tais como: índices de elementos, elementos vizinhos etc.

Para a implementação da metodologia proposta decidiu-se pela adoção da estrutura de classes proposta na Figura (6.3).

São implementadas três classes:

- *TPZAdaptMesh*: classe responsável pelo gerenciamento do processo adaptativo;

Figura 6.3: Estrutura de Classes para Implementação da Estimação de Erro Localmente



- *TPZGeoCloneMesh*: implementa uma malha geométrica a partir de um *patch* de elementos de referência. A cada elemento desta classe corresponde um elemento da malha geométrica original. As referências à malha original são definidas e armazenadas nos objetos desta classe;

- *TPZCompCloneMesh*: implementa uma malha computacional, a partir da malha clone geométrica - *TPZGeoCloneMesh*. Além dos elementos deste objeto ter a mesma solução que os seus elementos de referência na malha computacional, também são implementadas condições de contorno nos bordos da malha onde, na malha original, existiam elementos vizinhos;

Além destas novas classes implementadas, também foi necessário a implementação de algumas funções na classe *TPZCompMesh*, de modo a possibilitar a obtenção dos elementos de referência para o clone copiar os materiais da malha e também a criação de uma cópia de uma malha computacional.

Funções de classes já implementadas foram atualizadas, de modo a atender os requisitos destas novas classes implementadas.

A justificativa pela adoção desta estrutura é tornar possível a implementação da metodologia descrita anteriormente, tendo por base as ferramentas que o ambiente PZ oferece.

Assim, fazendo um paralelo com a metodologia proposta, o objetivo é, através de um conjunto de elementos, que forme uma partição da malha, criar para cada um destes elementos um *patch*, formado pelo próprio elemento mais seus vizinhos. Este *patch* seria equivalente a malha coarse, no método base.

Seguindo a metodologia proposta, sobre este *patch* será realizado um refinamento *hp* uniforme para então tomar esta solução como a solução de referência no cálculo do erro.

A obtenção dos elementos de referência e dos *patches* através de funções incrementadas a classe *TPZCompMesh* tem objetivo claro: obter os *patches* que servem de base para o método.

Já a criação das três classes propostas está relacionado à estrutura do ambiente PZ onde os elementos e malhas no PZ tem sua implementação dividida em dois conjuntos básicos:

- geométricos: implementam os aspectos relacionados à topologia de malhas e elementos;
- computacional: implementa os aspectos e funções algébricas sobre estes elementos.

Todos os métodos de resolução do PZ estão ligados ao fornecimento de uma malha, contendo todas as suas informações, tais como dados de materiais, topologia e conectividade dos elementos e nós além das condições de contorno.

Assim, a intenção das implementação das classes *TPZGeoCloneMesh* e *TPZCompCloneMesh* é transformar o *patch* de elementos selecionados em uma nova malha, malha esta com características idênticas a malha original, excetuando-se que para os nós, arestas e faces limites desta malha serão impostas condições de contorno do tipo Dirichlet, com valor da variável de estado imposta como sendo o valor da solução originais nestes elementos.

Já a classe *TPZAdaptMesh* faz o gerenciamento do processo adaptativo, conforme será descrito na seqüência.

A descrição dos métodos implementados foi dividida em termos de classes, sendo para cada classes descritos as variáveis, funções e métodos da classes. A forma de apresentação das classes é feita de tal forma a apresentar os métodos da interface com o usuário para a sua implementação, ou seja, a apresentação é feita das classes e funções de alto nível (interface com o usuário) para as classes e funções de baixo nível (implementação).

6.2.1 Classe TPZAdaptMesh

Esta classe é responsável pelo gerenciamento do processo adaptativo. A interface da classe é simples, sendo necessário para a obtenção da malha adaptada seguir os seguintes passos:

1. Criar um objeto: isto é simples, pois o construtor padrão da classe não requer argumento algum, assim isto é feito da seguinte forma: *TPZAdaptMesh adapt*; No caso, *adapt* foi o objeto criado.

2. Definir a malha computacional sobre a qual quer se obter a malha adaptada. Isto é feito através da função *void TPZAdaptMesh::SetCompMesh(TPZCompMesh *mesh)*; Aqui a malha *mesh* foi passada para a função como a malha a ser adaptada.

3. Requerer o cálculo da malha adaptada: isto é feito através da função TPZCompMesh* *TPZAdaptMesh::GetAdaptMesh(REAL &error, REAL & truerror, TPZVec<REAL> &ervec, void(*f)(TPZVec<REAL> &loc, TPZVec<REAL> &val, TPZFMatrix &deriv), TPZVec<REAL> &truervec, TPZVec<REAL> &effect)*, sendo devolvida pela

função, além da malha adaptada, informações relativas ao erro estimado e ao erro real, caso se tenha a solução analítica para o problema analisado. Observe que, conforme será descrito na seqüência, todos os parâmetros passados para a função são parâmetros que retornarão informações.

Estas três linhas que são necessárias para a obtenção da malha adaptada pode ser vista como um exemplo de utilização da orientação a objetos para esconder a complexidade de códigos, complexidade esta que é melhor descrita na seqüência, com a descrição da classe.

Variáveis / Atributos

- *TPZCompMesh *fReference*: armazena a malha de referência, ou seja, a malha sobre a qual está se querendo obter a malha adaptada;
- *TPZStack < TPZGeoEl * > fGeoRef*: armazena a lista de ponteiros para os elementos de referência dos *patches*, sendo estes ponteiros referidos à malha de referência;
- *TPZStack < TPZGeoEl * > fPatch*: armazena os *patches* de todos os elementos sob a forma de uma lista contínua, como em um armazenamento de matriz *skyline*;
- *TPZStack < int > fPatchIndex*: armazena os índices de início dos elementos de cada *patch* em *fPatch*.
- *TPZStack < REAL > fElementError*: armazena o erro em cada elemento da malha original, após o seu cálculo;
- *TPZStack<TPZCompCloneMesh *> fCloneMesh*: lista com as malhas computacionais relativas a cada *patch*;
- *TPZStack <TPZCompMesh *> fFineCloneMesh:* lista com as malhas computacionais uniformemente refinadas em *hp*;

Funções e Métodos

*void SetCompMesh(TPZCompMesh * mesh)*

Define a malha sobre a qual está sendo feito o processo adaptativo, inicializando todas as estruturas de dados.

*TPZCompMesh * GetAdaptedMesh(REAL &error, REAL &truerror, TPZVec<REAL> &ervec, void (*f)(TPZVec<REAL> &loc,TPZVec<REAL> &val,TPZFMatrix &deriv), TPZVec<REAL> &truervec, TPZVec<REAL> &effect)*

Os parâmetros informados ao método são:

- *error*: irá retornar o erro estimado para malha, com a utilização dos padrões de refinamento calculados;

- *truerror*: irá retornar o erro real devido a utilização do padrão de refinamento caso se disponibilize uma função com a solução analítica para o problema em questão.
- *ervec*: neste vetor será retornado o erro estimado para cada elemento da malha original;
- *(*f)(loc,val,deriv)*: esta função indica a solução analítica para o problema, caso esta não seja conhecida basta ser passado um nulo (0);
- *truervec*: caso se tenha a solução analítica, este vetor retornará o erro real em cada elemento;
- *effect*: vetor que retornará a efetividade do estimador de erros em cada elemento, caso tenha sido informada a solução analítica.

Este método é o responsável pela execução do processo adaptativo. Aqui são realizadas as seguintes tarefas:

1. Obtenção da lista de elementos de referência para o *patch* na malha original: isto é feito através da função *void GetReferenceElements()*, a descrição desta função é feita posteriormente;
2. Obtenção dos *patches* relativos aos elementos de referência encontrados. Isto é feito através da função *void BuidReferencePatch()*;
3. Criação das malhas clone. Os *patches* obtidos anteriormente são utilizados na criação de malhas clone. São criadas na seqüência a malha clone geométrica, sendo esta utilizada na criação da malha clone computacional. Somente são armazenadas como variáveis da classe as malhas clone computacionais, pois estas guardam referências para as malhas clone geométricas. A função responsável por estas operações é *void CreateClones()*;
4. Para cada malha clone:
   (a) cria-se uma malha uniformemente refinada, através da função *TPZCompMesh * TPZCompCloneMesh::UnifomlyRefineMesh()*, função esta da classe *TPZCompCloneMesh* que retorna uma malha proveniente do refinamento uniforme da malha analisada. A descrição desta classe, bem como da função será feita posteriormente;
   (b) calcula-se o erro entre a solução da malha uniformemente refinada e da malha clone original. Isto é feito através da função *TPZCompMesh * TPZCompCloneMesh::MeshError (TPZCompMesh *fine, TPZVec<REAL> & erro, void (*f), TPZVec<REAL> &truervec)*. Função esta que calcula o erro apenas no elemento de referência do clone e em seus filhos. Caso se tenha a função analítica, o erro real também será calculado.

5. Após ter se calculado o erro, devido ao refinamento uniforme, em todas as malhas clone, teremos que o vetor de erros - *fElementError*, estará completamente preenchido, tendo sido completada a estimativa de erro local. Ressalta-se novamente que o vetor de erros tem sua indexação relativa aos elementos da malha original, sendo na execução do método procurado o índice deste vetor ao qual corresponde o erro calculado através dos clones.
   A partir deste ponto, a implementação base foi adaptada, de modo a operar também com as malhas clone aqui implementadas.

6. Da mesma forma que no método base, o vetor de erros é ordenado, sendo então definidos os elementos com erro significativo que terão o seu padrão de refinamento *hp* analisado. A definição dos elementos significativos seguiu o mesmo raciocínio feito no método base, ou seja, são analisados os padrões dos elementos de maior erro, de tal forma que sejam analisados elementos cuja soma total do erro seja de, no mínimo, 65% do erro total estimado.

7. Tendo o erro mínimo para análise, definido acima, o próximo passo é a análise do padrão de refinamento de cada elemento, isto é feito através da utilização das malhas clone, em função de termos a solução uniformemente refinada para estas malhas, armazenadas em *fFineCloneMesh*. O cálculo do padrão de refinamento é feito através do método *void TPZCompCloneMesh::ApplyRefPattern(REAL minerror, TPZVec<REAL> error , TPZCompMesh &fine, TPZStack<TPZGeoEl *> gelstack, TPZStack<int> &porder)*. O método será melhor descrito posteriormente, bastando atentar para o fato de que no método serão preenchidos os padrões de refinamento de cada elemento da malha original, através das listas *gelstack* e *porder*, as quais contém os elementos geométricos e ordem de polinômio que deverão conter a malha adaptada;

8. Com os padrões de refinamento *hp* definidos, a função *TPZCompMesh * TPZAdaptMesh:: CreateCompMesh (fReference,gelstack,porder)*, irá se encarregar de definir a malha adaptada baseada na malha original e nos padrões de refinamento anteriormente calculados.

O diagrama de seqüência da Figura (6.4) mostra, simplificadamente, o método.

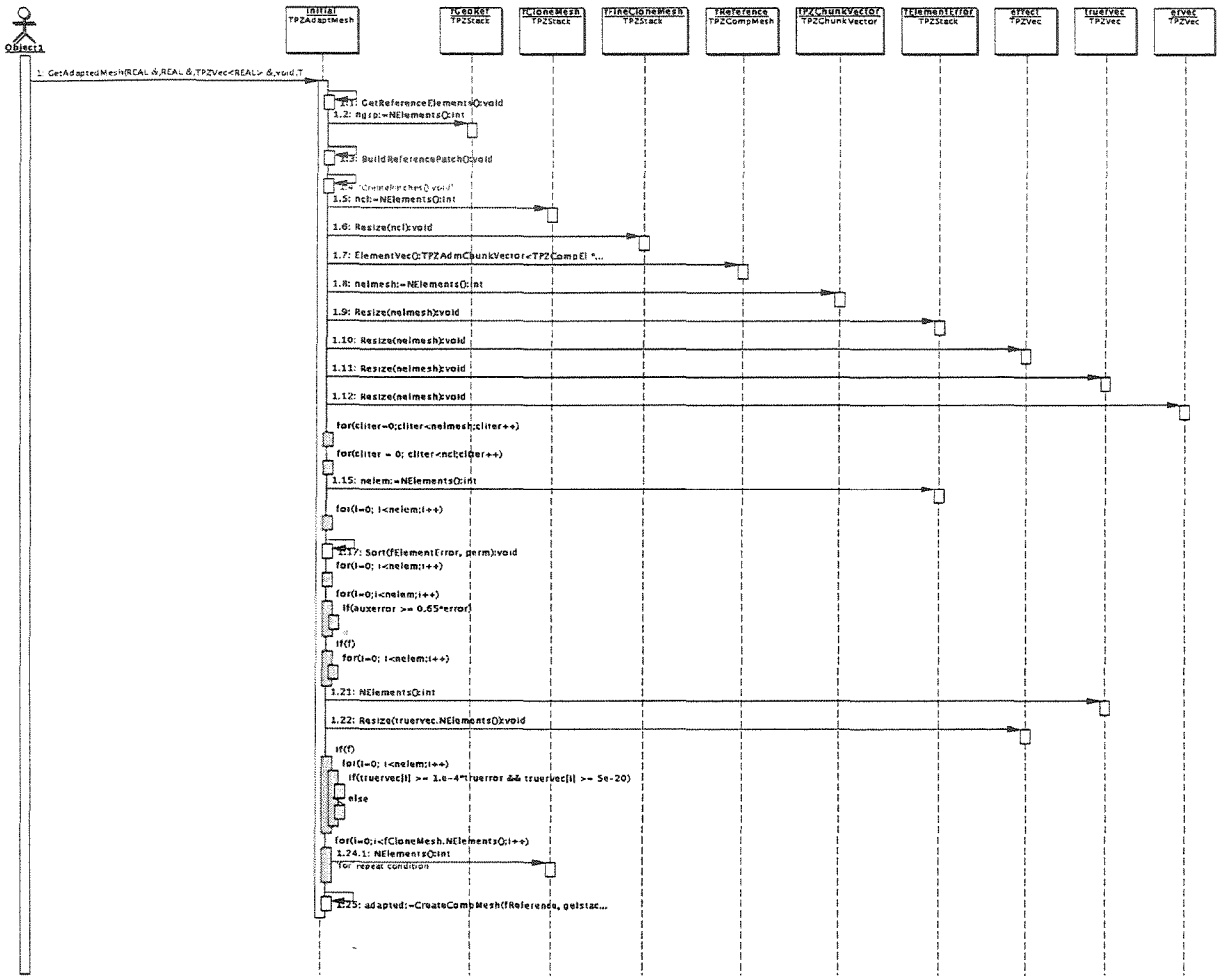
void GetReferenceElements()

Neste método são obtidos os elementos de referência para os *patches* e, tendo-se estes são redimensionados os vetores para as malhas clone e para as malhas clone uniformemente refinadas.

Basicamente o método realiza a chamada da função *TPZCompMesh::GetRefPatches (TPZStack<TPZGeoEl *> gelref)* sobre a malha de referência - *fReference*, passando como argumento para a função o vetor de elementos de referência para os *patches* - *fGeoRef*.

Dentro desta função, a qual será descrita posteriormente, são realizadas todas as operações para a obtenção deste vetor.

Figura 6.4: Método *GetAdatptedMesh*



void BuildReferencePatch()

Aqui são obtidos os *patches*, com base nos elementos de referência calculados anteriormente.

O processo aqui utilizado consiste em criar uma malha computacional, cujos elementos computacionais são criados a partir do vetor de elementos de referência para o *patch*. Com isto, teremos uma malha apenas com os elementos de referência.

No ambiente PZ, cada elemento é capaz de obter a lista de seus vizinhos, isto é possível devido a utilização de grafos de elementos para nós e vice-versa. Resumidamente, o elemento sabe quais são seus nós e, através do grafo de nós para elementos, os nós sabem a quais elementos ele pertence.

Desta forma, criando-se esta malha auxiliar é possível obter os *patches* de cada elemento da lista de elementos de referência.

O processo acima descrito é implementado da seguinte forma:

1. Cria-se uma malha computacional auxiliar, com base na malha geométrica associada a malha computacional de referência - *fReference*;

2. Desassocia-se a malha geométrica da malha computacional de referência e a associa à malha computacional auxiliar;

3. Criam-se elementos computacionais na malha auxiliar, com base nos elementos de referência para o *patch*;

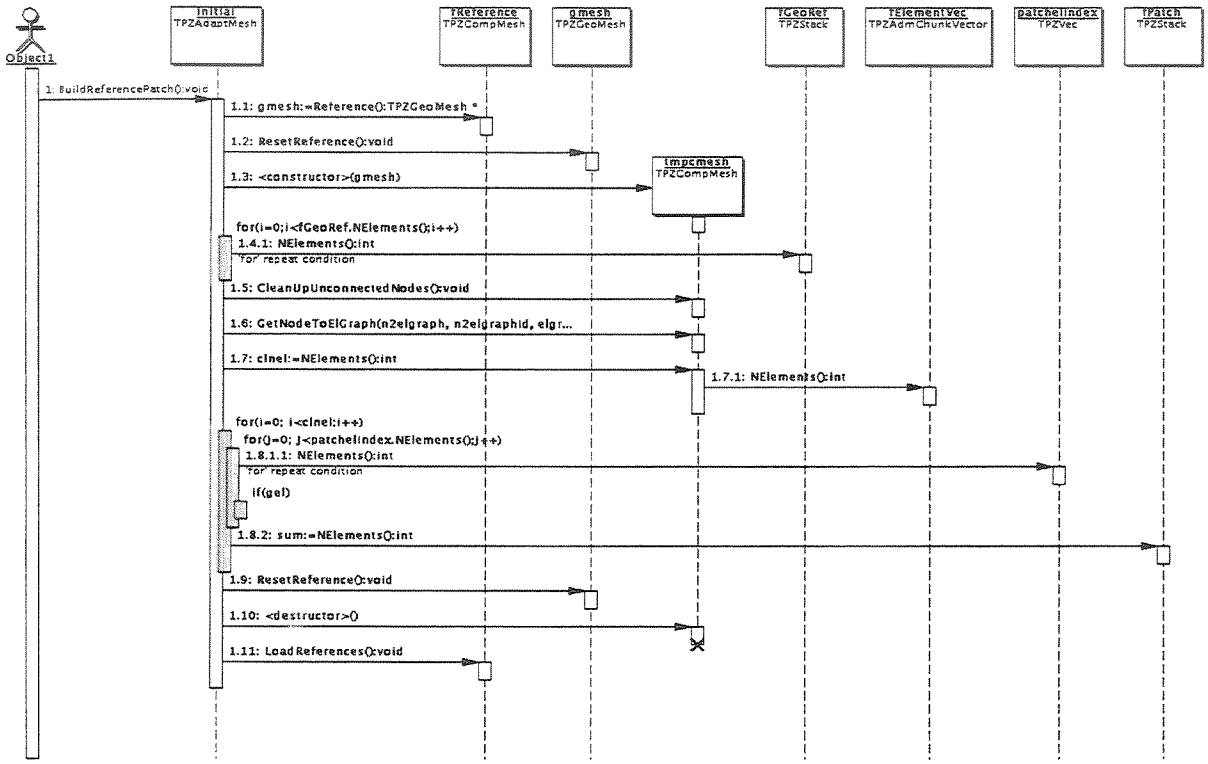
4. Obtém-se o grafo de nós para elementos da malha - método void *TPZCompMesh::GetNodToElGrap (TPZVec<int> n2elgraph, TPZVec<int> n2elgraphid, TPZStack <int> elgraph, TPZStack<int>elgraphindex)*. Todos os argumentos listados são argumentos que retornam valores, não sendo necessário nenhum cálculo prévio;

5. Tendo-se os grafos de nós para elementos, o passo seguinte é a obtenção do *patch* de cada elemento, sendo isto implementado na função *void TPZCompMesh::GetElementPatch(TPZVec<in n2elgraph, TPZVec<int> n2elgraphid, TPZStack <int> elgraph, TPZStack<int>elgraphindex,int elindex, TPZStack <int> patchelindex)*, onde são passados os dados dos grafos, obtidos no passo anterior, o índice do elemento cujo *patch* deseja-se obter e uma lista onde serão retornados os elementos componentes do *patch*.

6. Os elementos obtidos anteriormente são incluídos na lista de *patches* - *fPatch*. Preenche-se no vetor *fPatchIndex* o índice de início dos elementos do próximo *patch* a ser verificado. Após percorrer todos os elementos de referência do *patch*, teremos a lista dos elementos de cada *patch*.

7. Remove-se da malha auxiliar a referência para a malha geométrica e redefine-se esta referência para a malha computacional de referência - *fReference*.

Figura 6.5: *BuildReferencePatch*



O método é ilustrado pelo diagrama da Figura (6.5).

void CreateClones()

Com base nos *patches*, obtidos anteriormente, são criadas as respectivas malhas clone.

Como já mencionado anteriormente, uma malha no PZ tem uma representação geométrica e uma computacional, sendo a segunda gerada com base na primeira.

Desta forma o primeiro passo é a criação da malha clone geométrica - *TPZGeoCloneMesh(TPZGeoMesh *gmesh)*, sendo passado como argumento para a classe a malha geométrica relativa a malha de referência (*TPZGeoMesh * gmesh = fReference->Reference()*). Os aspectos envolvidos na criação desta malha serão abordados posteriormente.

Após a criação do objeto malha clone geométrica, há a necessidade de se informar quais são os seus elementos, sendo isto feito através do método *void TPZGeoCloneMesh::SetElements(TPZStack <TPZGeoEl *> patch, TPZGeoEl *refgel)*. Os argumentos passados ao método são o *patch* e o elemento de referência a que a malha referencia.

De maneira básica, a informação relativa ao elemento de referência servirá para indicar em quais elementos deve ser calculado o erro, pois os elementos vizinhos a este também estão contidos em algum *patch*, na condição de elementos de referência ou filhos deste, devendo seu erro ser calculado naquela malha clone.

Tendo a malha clone geométrica, o passo seguinte é a criação de um objeto malha clone computacional, sendo esta criação feita através do construtor *TPZCompCloneMesh:: TPZCompCloneMesh (TPZGeoCloneMesh *gmesh, TPZCompMesh *refmesh)*, sendo *gmesh* o ponteiro para a malha clone geométrica e *refmesh* o ponteiro para a malha computacional de referência.

Sobre o objeto malha clone computacional é executado *TPZCompCloneMesh::AutoBuild()*, método cujas responsável pela "montagem" propriamente dita da malha, pela transferência da solução da malha original para a malha clone e também pela criação das condições de contorno nas bordas da malha clone nas quais existem elementos na malha original. Este método, fundamental no processo, tem sua descrição pormenorizada realizada posteriormente.

Com a malha clone computacional criada e tendo todas as suas características já definidas, o passo seguinte é a inserção desta malha no vetor de malhas clone - *fCloneMesh*.

void Sort(TPZVec<REAL> &vec, TPZVec<int> &perm)

Este método é um simples ordenador, de modo a ordenar de maneira decrescente o vetor de erros - *vec*, fornecendo os índices de permutação - *perm*.

TPZCompMesh CreateCompMesh (TPZCompMesh *mesh, TPZVec<TPZGeoEl *> &gelstack, TPZVec<int> &porders)*

Esta função tem por objetivo criar uma malha adaptada, a qual será retornada pela função, com base em uma malha original e nos vetores de elementos geométricos, os quais indicam o refinamento h ou não, e o refinamento p de cada elemento geométrico indicado.

Simplificadamente, o método copia o vetor de materiais da malha original e utiliza a sua malha geométrica de referência para criar uma nova malha computacional, sendo nesta definido o vetor de materiais igual ao vetor da malha original - *mesh*. Note que antes da

criação da nova malha computacional, a malha geométrica tem sua referência para a malha computacional anulada. Isto é feito para que a malha geométrica referencie a malha adaptada, após sua criação.

O passo seguinte é a criação dos elementos computacionais da malha com base nos elementos geométricos - *gelstack* e na ordem dos polinômios - *porders*.

TPZCompMesh

Como já descrito anteriormente, esta classe já estava implementada, sendo aqui descritos os métodos adicionados à classe. A documentação completa desta classe pode ser consultada no endereço internet: `http://labmec.fec.unicamp.br/~pz`.

*void GetRefPatches(TPZStack<TPZGeoEl *> &grpatch)* - Identificação dos Elementos de Referência para o *Patch*

Este método é responsável pela identificação de todos os elementos que servirão de elementos de referência para a criação de *patches*.

Argumentos

- *TPZStack<TPZGeoEl *> &grpatch*: referência para lista de ponteiros para elementos geométricos. Esta lista será preenchida durante a execução do método

Operações

Conforme descrito anteriormente, no algoritmo 2d, sendo aqui mostrado como as operações anteriormente descritas são implementadas.

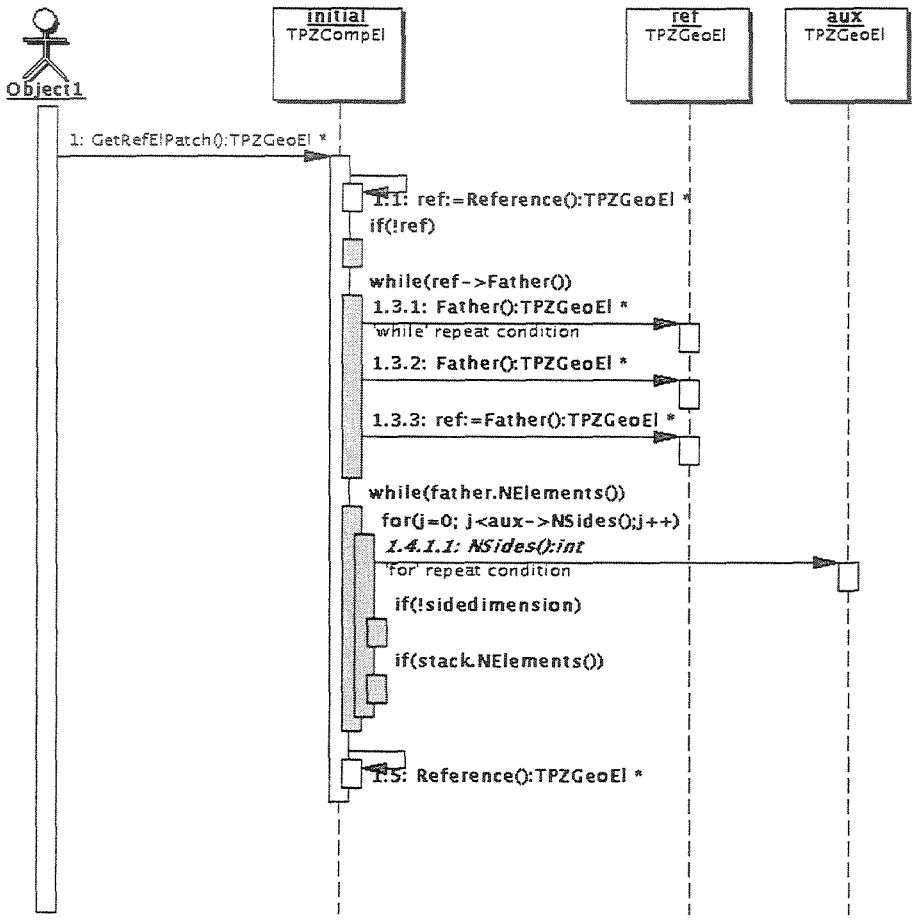
Para facilitar a visualização do texto, foi inserido o diagrama de seqüência, mostrado na Figura (6.6), o qual mostra a seqüência de operações para a obtenção do elemento de referência de cada elemento da malha computacional.

Assim, o processo na malha computacional, consiste em percorrer todos os elementos computacionais e chamar o método *void TPZCompEl::GetRefElPatch(TPZGeoEl *ref)*.

Neste método, temos:

- Atributos:
  - *TPZGeoEl * ref*: retorna um ponteiro para o elemento geométrico identificado como sendo o elemento de referência para o elemento dado;
- Operações: conforme descrito na metodologia, o método consiste em:
  - preenchimento de uma lista de ancestrais do elemento de referência do elemento computacional analisado. A obtenção do ancestral de um elemento geométrico é feita através da utilização da função *TPZGeoEl * Father()*;

Figura 6.6: Obtenção do Elementos de Referência do *Patch*



- percorrer a lista de ancestrais indo do último elemento inserido (elemento da malha inicial, antes de qualquer refinamento), para o elemento de referência do elemento computacional analisado. Para cada elemento desta lista:
  * obter a lista de vizinhos de mesma ordem do elemento em análise,
  * caso a lista de vizinhos não seja nula este será o elemento de referência.

Tendo o elemento de referência do elemento, o próximo passo é verificar se este já consta da lista de elementos geométricos, passada como parâmetro para o método. Em caso negativo o elemento obtido é inserido na lista.

Desta forma, após todos os elementos da malha computacional terem sido percorridos, a lista de elementos passada como parâmetro conterá a lista dos elementos de referência para a criação de *patches*.

void GetElementPatch(TPZVec<int> nodtoelgraph, TPZVec<int> nodtoelgraphindex, TPZStack<int> &elgraph, TPZVec<int> &elgraphindex,int elind ,TPZStack<int> &patch)

Este método é o responsável pela identificação do *patch* de um dado elemento.

Os argumentos são os seguintes:

- *nodtoelgraph & nodetoelgraphidex*: grafo de nós para elementos;
- *elegraph & elgraphindex*: grafo de elementos para nós;
- *elind*: índice no vetor de elementos da malha, indicando o elemento sobre o qual se quer o *patch*;
- *patch*: lista na qual será devolvida a lista de elementos componentes do *patch*;

Este método consiste em: dados os grafos de nós para elementos e de elementos para nós, obter para cada nó do elemento a lista de elementos conectados a este nó, sendo estes inseridos na lista do *patch*, caso ainda não estejam nela.

*TPZCompMesh * Clone()*

Este método cria uma cópia da malha sobre a qual a função é aplicada.

As operações necessárias são:

1. Copiar o vetor de materiais;
2. Remover a referência da malha geométrica de referência (a malha a ser criada necessita desta referência durante sua criação);
3. Criar uma cópia de todos os elementos computacionais e definir a ordem de refinamento p da cópia igual ao do elemento original;
4. Remover a referência da malha criada e redefinir a referência para a malha de referência;

5. Copiar a solução da malha de referência para a cópia, bloco a bloco.

Ao final do processo ter-se-á uma cópia da malha atual devendo ser ressaltado que neste ponto malha geométrica está referenciando a malha original.

6.2.2 Malha Clone Geométrica

Esta classe é responsável pela representação de uma malha geométrica baseada em um *patch* de elementos de uma malha de referencia.

A principal característica de um objeto desta classe é o mapeamento de seus elementos e nós para os elementos e nós da malha geométrica de referência e também, um elemento geométrico de referência, sendo neste elemento e em seus filhos aplicadas as operações de interesse no processo adaptativo.

A implementação baseou-se na derivação da classe *TPZGeoMesh*, sendo acrescidas variáveis e funções conforme será descrito na seqüência.

Variáveis / Atributos

- *TPZGeoMesh *fGeoReference:* malha geométrica de referência;
- *TPZAVLMap <int,int> fMapNodes*: mapeia os índices dos nós no vetor de nós da malha original para os índices dos nós da malha clone;
- *TPZAVLMap <TPZGeoEl *,TPZGeoEl *> fMapElements*: mapeia os ponteiros dos elementos geométricos da malha de referência para os ponteiros dos elementos geométricos da malha clone;
- *TPZStack<TPZGeoEl *> fReferenceElement*: faz o mapeamento de elementos inverso, ou seja, dado do índice do elemento geométrico na malha clone é retornado o índice do elemento geométrico na malha original;
- *TPZStack<TPZGeoEl *> fPatchReferenceElements*: neste vetor são armazenados o elemento geométrico de referência para a malha e também os seus filhos;
- *TPZStack<TPZGeoEl *> fPatchElements*: são armazenados os ponteiros para os elementos fornecidos no *patch*;
- *TPZGeoEl* fGeoElRef*: armazena o elemento da malha clone referente ao elemento de referência do *patch*;

Funções e Métodos

*TPZGeoClone::TPZGeoCloneMesh(TPZGeoMesh *mesh)*

Construtor da classe. Cria um objeto malha clone geométrica com base em uma malha geométrica de referência (*fGeoReference=mesh*);

virtual ~TPZGeoCloneMesh()

Destrutor simples.

*void SetElements(TPZStack<TPZGeoEl *> &patch, TPZGeoEl *ref)*

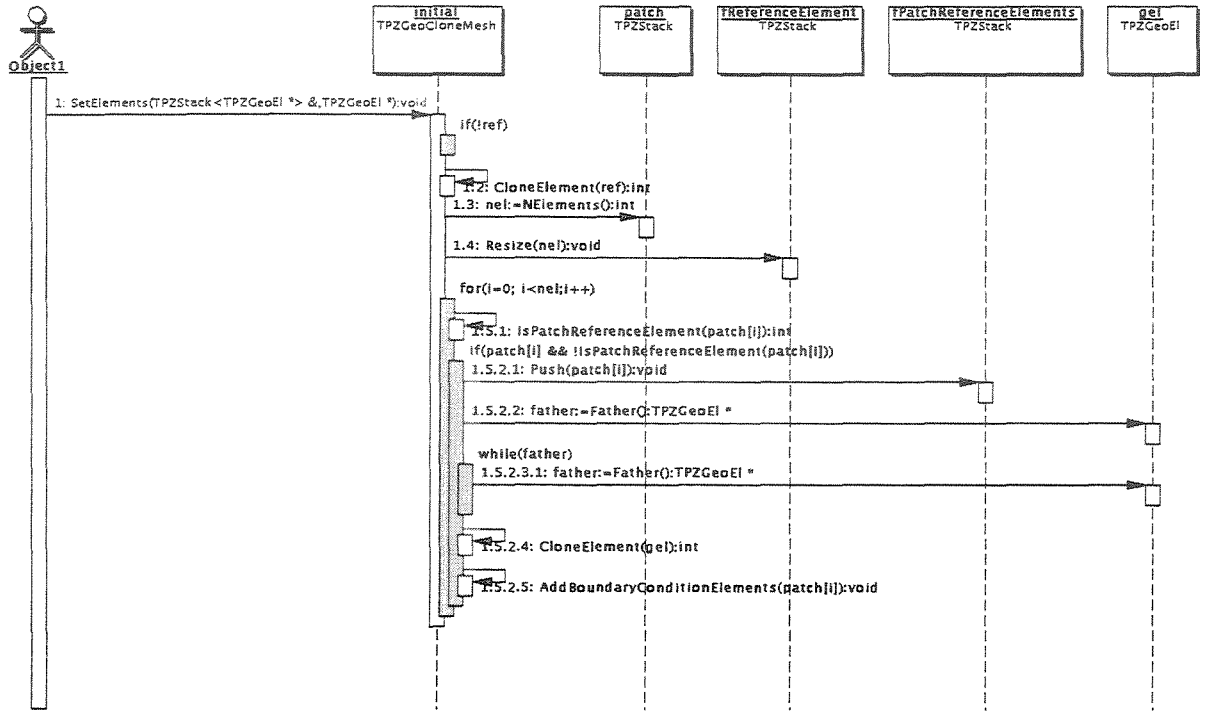
Neste método é que ocorre a criação de todos os componentes da malha, sendo passados como parâmetros:

- *patch*: *patch* de elementos da malha de referência que comporão a malha
- *ref*: elemento geométrico de referência do *patch* fornecido;

Com base nestas duas informações serão realizadas as seguintes operações:

1. Criação do clone do elemento de referência do *patch*: realizado através da função *int CloneElement(TPZGeoEl *orggel)*, a qual retorna o índice do elemento criado. A descrição desta função será feita posteriormente;
2. Definição do elemento de referência da malha clone, sendo este o resultado da função acima, ou seja: *fGeoElRef=fMapElments[orggel]*;
3. Para todos os elementos do *patch*:
   (a) Verificar se o elemento já consta da lista de elementos clonados ou se é nulo, em caso afirmativo passa para o próximo elemento e em caso negativo continua-se,
   (b) Adicionar o elemento do *patch* à lista de elementos *fPatchElements* ,
   (c) Procura-se o elemento pai referente à malha inicial do elemento de *patch*, isto é feito através da utilização da função *TPZGeoEl*::Father()*, sobre o elemento, de maneira recursiva, até que a função retorne nulo, o que indicará que o elemento não tem pai e, desta forma, é um elemento da malha original;
   (d) Clona-se o elemento pai da malha original, função *int CloneElement(TPZGeoEl *orggel)*;
   (e) Verifica-se a necessidade de adicionar condições de contorno ao elemento. Isso ocorre caso o elemento seja um elemento de bordo da malha clone e na malha original este elemento apresente elementos vizinhos. Isto é feito através do método *void TPZGeoCloneMesh::AddBoundaryConditionElements(TPZGeoEl *el)*.

Figura 6.7: *SetElements*



Nos ítens 3.(b) e 3.(c), destaca-se o fato de que a malha geométrica de clone tem sempre os elementos relativos à malha inicial em sua composição, entretanto, apenas o elemento que realmente faz parte do *patch* é inserido no vetor de elementos do *patch*, vetor este cujos elementos são aqueles passíveis de qualquer operação nesta malha.

O diagrama de seqüência da Figura (6.7) ilustra a implementação deste código.

*int CloneElement(TPZGeoEl *orggel)*

Esta função realiza as operações necessárias à clonagem de um elemento geométrico, preenchendo de maneira adequada as variáveis de mapeamento da classe.

O argumento da classe é um ponteiro para o elemento geométrico da malha original - *orggel*.

As operações realizadas dentro desta função são as seguintes:

1. Verificar se o elemento *orggel* já foi clonado, sendo isto feito através da função *int TPZGeoCloneMesh::HasElement(TPZGeoEl * orggel)*. Esta função verifica se o ponteiro *orggel* já consta da lista de elementos mapeados.
   Caso o elemento já conste da lista de elementos mapeados a função retorna o índice do elemento clonado no vetor de elementos da malha clone.

2. Em caso de necessidade de clonar o elemento, o primeiro passo é descobrir o tipo de elemento que está sendo analisado. Isto é feito através da função *TPZGeoEl * TPZGeoCloneMesh::InitializeClone(TPZGeoEl* orgel)*. Esta função retornará um ponteiro para o tipo correto de elemento geométrico, sendo sua descrição feita posteriormente.

3. Tendo-se o tipo de elemento passa-se a verificar se os nós do elemento já foram clonados. Em caso afirmativo basta obter o índice do nó através do mapeamento entre os nós da malha original e os nós da malha clone - *fMapNodes[orgnodindex]*. Em caso negativo, é chamada a função *int TPZGeoCloneMesh::CloneNode(int orgnodeindex)*. A função *CloneNode* será descrita posteriormente.
   Tendo-se os índices dos nós clonados, define-se esta informação no elemento através do método void *TPZGeoEl::SetNodeIndex(int id, int nodeindex)*;

4. Preenche-se as listas de mapeamento de elementos da malha original para a malha clone e vice-versa.

5. Percorre-se todos os lados do elemento clonado e ajustam-se os dados de vizinhança de cada lado do elemento clonado - *void TPZGeoElSide::SetConnectivity (fMapElements [neighbour.Element()])*.

6. Percorre-se a lista de todos os subelementos do elemento original e para cada um destes elementos é chamada a função *CloneElement(subel)*, sendo para o elemento resultante da clonagem feitas as seguintes operações:

   (a) *void TPZGeoEl::SetSubElement(TPZGeoEl *sonofclonegel)* - definir o elemento clonado como um subelemento do elemento em análise;

   (b) *void TPZGeoEl::SetFather(TPZGeoEl *clonegel)* - definir o elemento em análise como elemento pai do subelemento clonado

7. Ao final do processo é retornado o índice do elemento clonado no vetor de elementos computacionais da malha clone.

De modo a facilitar a visualização da função, seu diagrama de seqüência é mostrado na Figura (6.8).

*int HasElement(TPZGeoEl *el)*

Esta função serve para verificar se o elemento passado como argumento da função - *el*, que pertence a malha original, já foi clonado ou não.

A verificação é feita através de uma busca com árvore binária, com a utilização de iteradores, próprios de funções de mapeamento.

*TPZGeoEl * InitializeClone(TPZGeoEl* orgel)*

O objetivo desta função é obter o tipo de elemento correspondente a *orgel*. Isto é feito através da criação de ponteiros para os diversos tipos de elementos componentes do PZ, sendo retornado o primeiro ponteiro não nulo obtido.

Figura 6.8: *CloneElement*

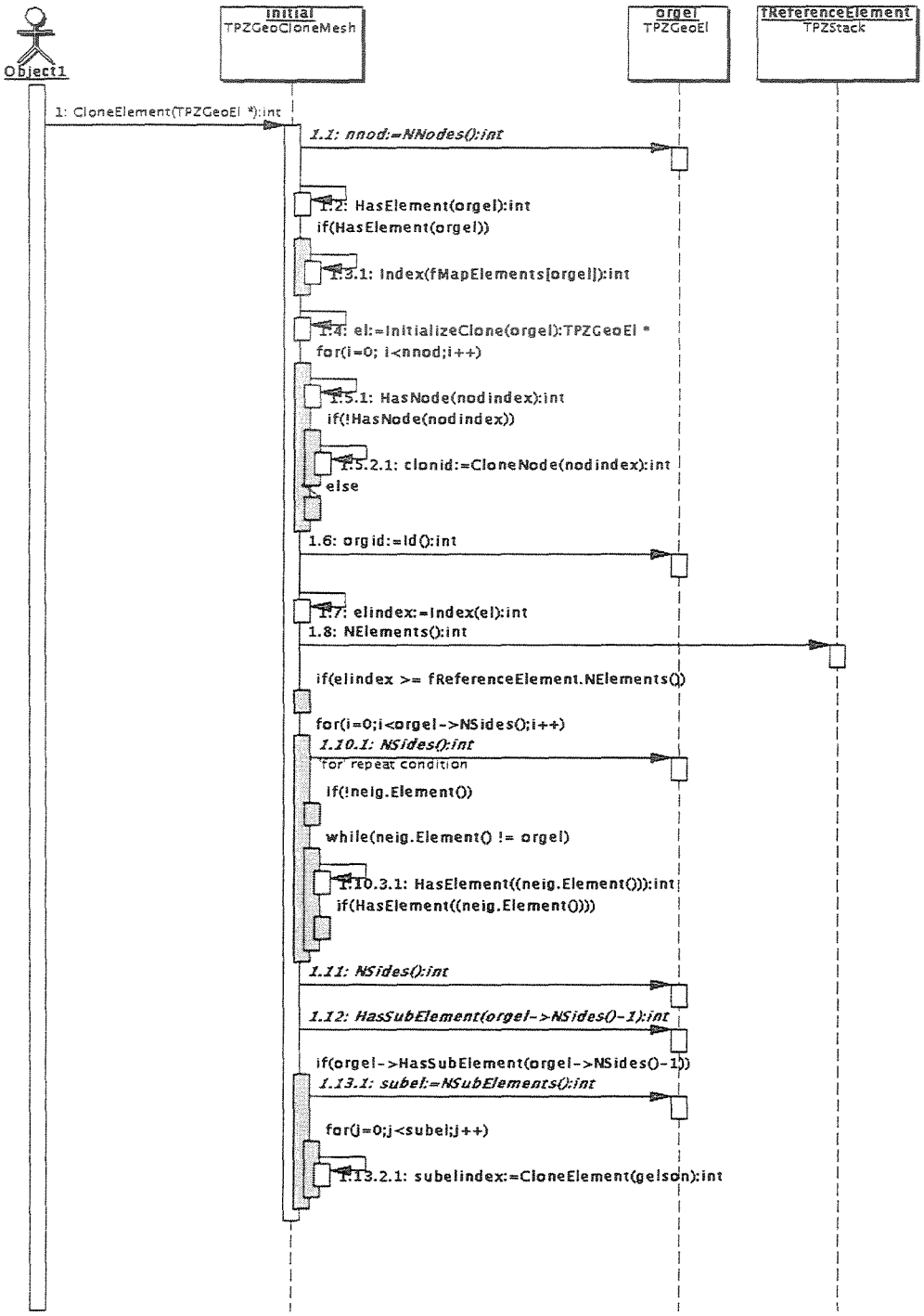
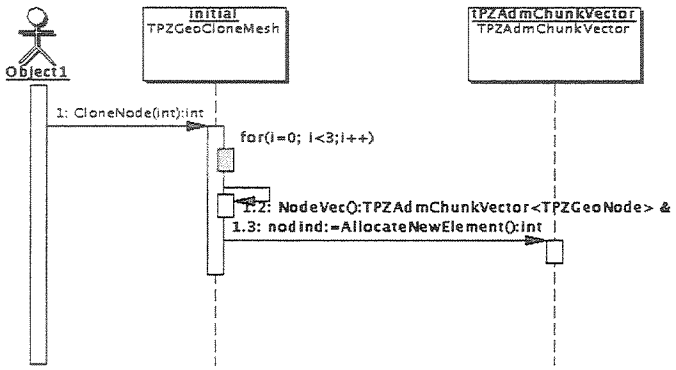


Figura 6.9: *CloneNode*



Isto pode ser feito devido à forma como é gerenciada a derivação na linguagem C++, no caso de tentar-se criar um ponteiro para um objeto derivado de uma classe , passando como argumento desta operação um objeto, também derivado da mesma classe, não coincidente ao tipo de objeto que está se tentando criar, obtém-se um ponteiro nulo, pois os dados de uma classe e outra não são condizentes, de modo a possibilitar tal operação.

Sendo descoberto o tipo de elemento já é feita a inicialização dos índices de nós do elemento.

int CloneNode(int nodindex)

Da mesma forma que para a função *CloneElement*, aqui são criados novos nós, sendo preenchidos as variáveis de mapeamento entre os nós da malha clone e da malha original.

Um aspecto a ser destacado é que os identificadores dos nós - *NodeIds*, tem seu valor copiado da malha original para a malha clone.

Com isto garante-se que a orientação das funções de forma na malha clone é a mesma da malha original, o que implica, dentre outros aspectos, que os blocos da solução da malha original coincide com os blocos da malha clone.

Esta função é mostrada através do diagrama de seqüência da Figura (6.9).

HasNode(int nodeindex)

Da mesma forma que para a a função *HasElement*, aqui é feita a verificação se um dado nó já consta da lista de nós da malha clonado, sendo isto feito através de uma busca com árvore binária.

TPZGeoEl ReferenceElement(int i)*

Retorna o elemento de referência do elemento i caso este exista. Observe que o elemento i pode ser objeto de um refinamento de um elemento qualquer, neste caso o elemento de referência é o elemento ancestral definido quando da criação da malha.

void Print (ostream & out)

Imprime as variáveis da classe no dispositivo indicado por *out*.

*int Index(TPZGeoEl *gel)*

Retorna o índice do elemento, passado através de *gel*, no vetor de elementos da malha clone. Este método implica em uma busca linear.

*TPZGeoEl * GetMeshReferenceElement()*

Retorna o elemento de referência do *patch* - *fGeoElRef*.

*int IsPatchSon(TPZGeoEl *gel)*

Função que retorna zero caso o elemento informado ou qualquer um de seus ancestrais não conste da lista de elementos do *patch* original ou um em caso afirmativo.

Como destacado anteriormente, nem todos os elementos da malha clone são elementos passados inicialmente na lista do *patch*. Esta função visa justamente esta verificação.

static int main()

Função com rotinas e testes de depuração do código implementado.

6.2.3 Malha Clone Computacional

Esta classe é responsável pela representação de uma malha computacional, baseada em um *patch* de elementos de uma malha de referência dado através de uma malha clone geométrica - *TPZGeoCloneMesh*.

A principal característica de um objeto desta classe é o mapeamento de seus elementos e nós para os elementos e nós da malha computacional de referência e também, a representação do *patch* através de uma malha. Sobre esta malha é transferida a solução da malha original e nos contornos desta malha, onde na malha original existiam elementos, são colocadas condições de contorno, de modo a fixar a solução original.

A implementação baseou-se na derivação da classe *TPZCompMesh*, sendo acrescidas variáveis e funções conforme será descrito na seqüência.

Variáveis / Atributos

- *TPZCompMesh * fCloneReference*: malha computacional de referência / original;
- *TPZAVLMap <int,int> fMapConnects*: mapeia as conectividades da malha original para a malha clone;
- *TPZStack <int> fOriginalConnects*: realiza o mapeamento inverso, ou seja dado o índice do elemento da malha clone é retornado o elemento da malha de referência.

Funções e Métodos

*TPZCompCloneMesh (TPZGeoCloneMesh * gr, TPZCompMesh * cmesh)*

Construtor da classe. Cria um objeto do tipo malha clone computacional sendo para tal passados os seguintes parâmetros:

- *gr* - malha clone geométrica que traz informações do *patch* de elementos a ser clonado, bem como informações relativas ao elemento de referência do *patch*;

- *cmesh*: malha computacional original a qual será utilizada para a obtenção de dados de materiais, solução e condições de contorno.

As operações realizadas no construtor são as seguintes:

1. Definir a malha geométrica associada: como esta é uma classe derivada de *TPZCompMesh*, é informado a malha geométrica necessária a criação de um objeto *TPZCompMesh*, ou seja : *TPZCompMesh(gr)*;

2. Definir a malha computacional de referência como sendo a malha fornecida: *fCloneReference = cmesh*;

3. Copiar o vetor de materiais da malha original para a malha clone.

~TPZCompCloneMesh()
Destrutor padrão.
virtual void AutoBuild()
Esta função é responsável pela montagem da malha computacional tendo por base a malha geométrica e o vetor de materiais. Nesta redefinição, são também realizadas as seguintes operações:

1. Criação dos elementos computacionais a partir da lista de elementos componentes do *patch*, dada pela malha clone geométrica. Para tal, para cada elemento da malha clone geométrica é chamada a função *int TPZGeoCloneMesh::IsPatchSon(TPZGeoEl *gel)*, já descrita.

2. Criação dos elementos computacionais para os elementos geométricos componentes do *patch*. Isto é feito através da função *TPZCompEl * TPZGeoEl::CreateCompEl(TPZCompM &mesh, int &index)*, sendo o primeiro argumento da função correspondente a malha onde será criado o elemento computacional, aqui foi passada a malha atual - **this*. O segundo argumento retornará o índice, no vetor de elementos computacionais da malha, do elemento criado.

3. O passo seguinte é o mapeamento das conectividades do elemento computacional criado. Para tal inicialmente é verificado a existência da conectividade do elemento de referência na variável de mapeamento, sendo isto realizado através da função *int TPZCompCloneMesh::HasConnect(int orgconindex)*. Caso essa conectividade ainda não faça parte da lista de conectividades mapeada, esta é então mapeada.

4. Tendo-se as conectividades mapeadas, para cada lado do elemento de referência, obtém-se a ordem p de refinamento do lado e utiliza esta ordem p para refinar o lado do elemento da malha clone.

5. Remove-se os nós não conectados a elementos - *void TPZCompMesh::CleanUpUnnconnectedNodes*

6. Inicializa-se a estrutura de dados para a solução;

7. Verifica-se a necessidade de criação de condições de contorno, incluindo as condições de bordo da malha clone, devido ao recorte do *patch* em relação a malha original. Isto é feito no método void *TPZCompCloneMesh::CreateCloneBC()*, o qual será descrito posteriormente.

8. Criam-se os elementos computacionais relativos às condições de contorno.

9. Cria-se a estrutura de blocos da malha - método *void TPZCompMesh::InitializeBlock()*.

10. Copiar a solução da malha de referência para a malha clone. Isto é feito através da cópia da solução bloco a bloco. Sendo obtido, para cada conectividade mapeada, a posição e tamanho do bloco associado, sendo este bloco copiado para a posição correta na malha clone.

Ao final do processo temos uma malha computacional com as características procuradas, ou seja constituída apenas pelos elementos do *patch*, com solução igual à solução da malha original e tendo seus contornos fixados com a solução da malha original. Este método é ilustrado pela Figura (6.10).

int HasConnect(int cnid)

Esta função verifica se um dado índice de uma conectividade na malha de referência já consta da lista de conectividades mapeadas.

void CreateCloneBC()

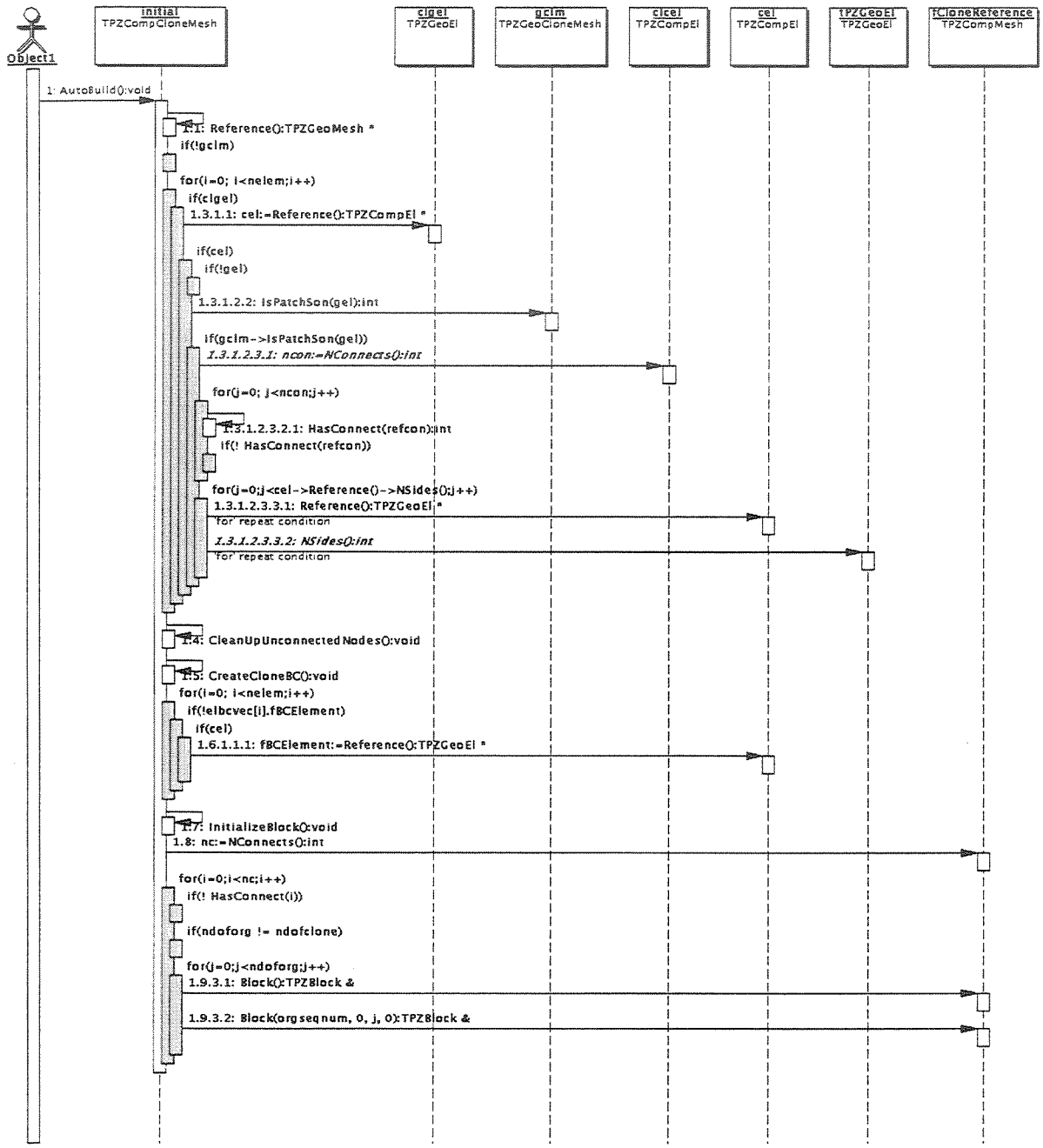
Este método é responsável pela criação das condições de contorno nos bordos da malha clone, onde na malha original existem elementos. Estas condições de contorno visam transmitir à malha clone a contribuição dos elementos que não compõem o *patch*.

A condição de contorno adotada é uma condição de Dirichlet, com valor da solução fixado igual a solução para os nós de interface na malha original.

Para ser iniciado o processo é necessário que todos os elementos da malha clone e suas respectivas conectividades estejam já definidas, incluindo a definição das referências para a malha original.

O cerne do método é verificar o número de elementos ligados a cada conectividade na malha original e na malha clone. Caso o número de conectividades na malha clone seja menor que na malha original isto indicará que esta conectividade é de bordo e desta forma deve-se identificar os elementos que contribuem na malha original e não contribuem na malha

Figura 6.10: *AutoBuild*



clone. A solução deste elementos é então utilizada como condição de contorno para esta conectividade.

A implementação desta operação é feita da seguinte forma:

1. Cria-se um material do tipo condição de contorno, o qual é inserido na malha. O valor da condição de contorno será fixada posteriormente.

2. Para cada elemento percorrer todos os lados do elemento e para cada lado:

   (a) Obter o lado em análise - *TPZGeoElSide side(int elindex, ins sideid)*;

   (b) Obter a conectividade ligada ao lado - *int TPZGeoElSide::ConnectIndex()*;

   (c) Identificar o número de elementos ligados a esta conectividade na malha clone, sendo isto feito através da utilização do vetor de conectividades da malha para obter o objeto *connect*, e sobre este objeto é aplicada a função *NElConnected()*;

   (d) Com o índice da conectividade na malha clone obtém-se o índice da conectividade na malha de referência - *int orgconid = fOriginalConnects [cloneconnectid]*;

   (e) Com o índice da conectividade na malha original é obtido o número de elementos conectados, da mesma forma que aquilo feito na malha clone;

   (f) Verificar a existência de elementos dependentes da conectividade analisada - função *int TPZConnect::HasDependency()*;

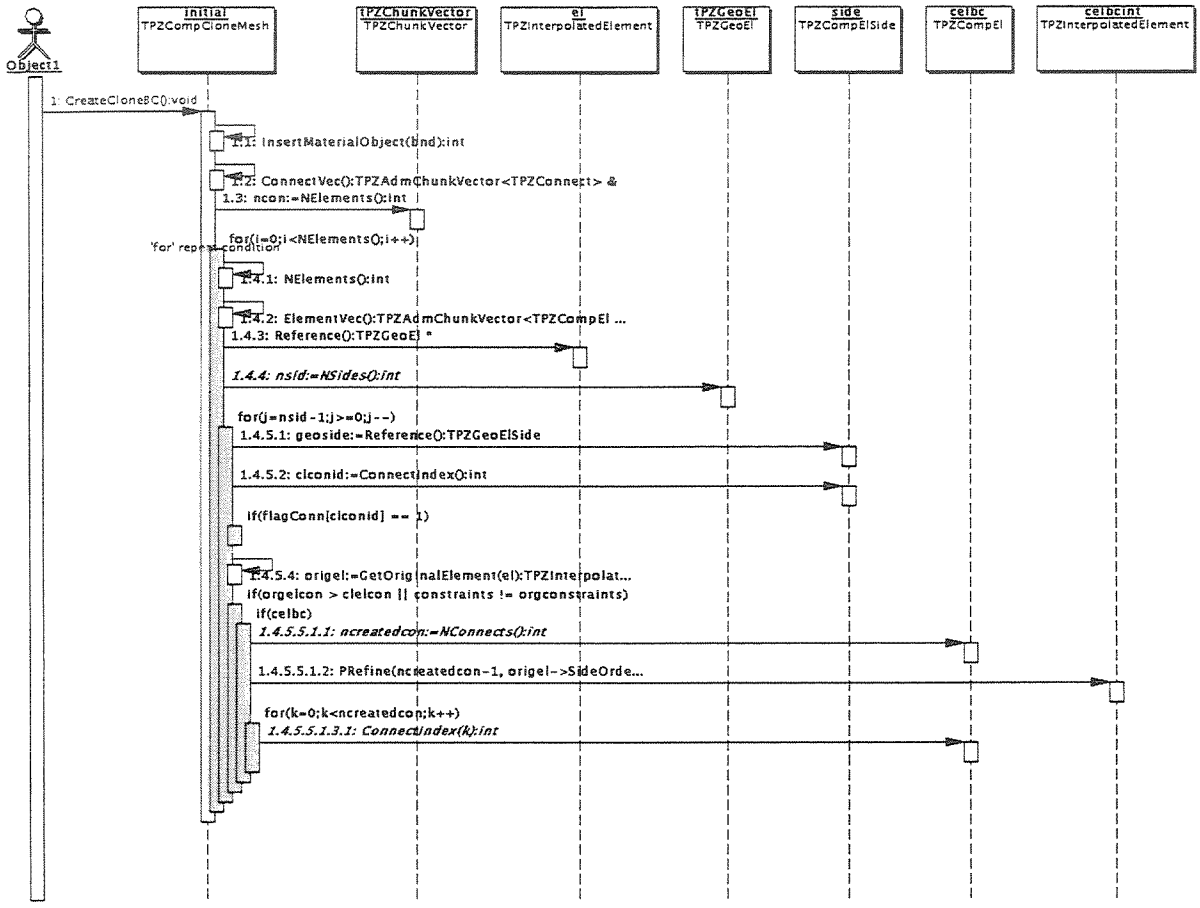
3. Caso o número de conectividades e de restrições seja igual no elemento da malha clone e no elemento da malha original, esta conectividade não necessita imposição de condições de contorno.

4. Em caso negativo, ou seja conectividades ou restrições em desacordo, há a necessidade de criação de um elemento condição de contorno - função *TPZCompEl * TPZGeoEl:: CreateBCCompEl (int sideid, int type, TPZCompMesh &mesh)*, sendo a função executado sobre o elemento geométrico associado ao elemento computacional em análise e o parâmetros são o lado do elemento no qual será criada a condição de contorno, o tipo de condição, que no caso por ser uma condição de malha clone foi convencionado seu tipo como sendo igual a -1000 e a malha computacional clone.

5. O elemento condição de contorno criado tem ainda sua ordem p compatibilizada, de modo a este elemento não restringir os elementos da malha clone.

6. Tendo sido percorridos todos os lados de todos os elementos as condições de contorno estarão todas ajustadas.

Figura 6.11: *CreateCloneBC*



O diagrama de seqüência da Figura (6.11) ilustra o método.

*TPZCompMesh * UniformlyRefineMesh()*

Este método retorna uma malha uniformemente refinada em *hp*, tendo por base a malha clone atual.

A implementação do código global já contemplava uma função de refinamento uniforme, servindo esta de base para a implementação aqui realizada.

As operações realizadas neste método são as seguintes:

1. Cria uma cópia da malha atual através da função *TPZCompMesh * TPZCompMesh::Clone()*. A implementação desta função já foi feita anteriormente.

2. Remove-se a referência da malha geométrica da malha clone e passa-se para a malha cópia. Isto é feito através da função *void TPZGeoMesh::ResetReference()*, na malha geométrica e na malha cópia é feita a chamada *void TPZCompMesh::LoadReferences()*.

3. Para cada elemento da malha cópia:

   (a) Obter a ordem de refinamento p do elemento em análise. Isto é feito a partir da utilização da função *int TPZInterpolatedElement::PreferredSideOrder(int side)*, sendo passado como argumento o maior lado do elemento. Isto é feito pelo fato de no ambiente PZ, a numeração dos lados começa pelas conectividades, passando para as arestas, faces e por último o volume, sendo a ordem do elemento em si a ordem do seu maior lado, volume no caso de um sólido por exemplo.

   (b) Realizar o refinamento do elemento através do método *void TPZCompEl::Divide(TPZCompE *el,TPZVec<int> &subelindex, int interpolatesolution)*, sendo passados como argumentos para esta função o elemento a refinar, um vetor que retorna os índices dos subelementos criados e o último parâmetro é uma chave para indicar se a solução do elemento original deve ou não ser interpolada para os subelementos - 0 não interpola e 1 interpola.

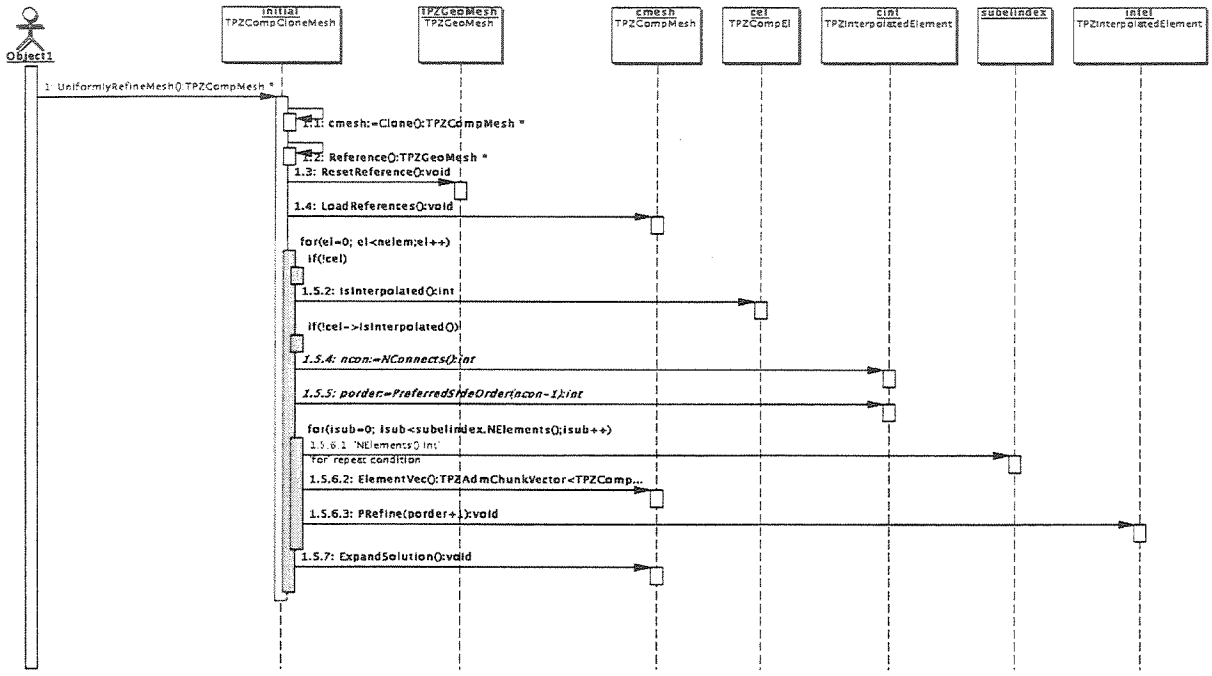
   (c) Refinar os subelementos em p, utilizando a ordem obtida anteriormente somando-se um. Isto é feito através do método *void TPZInterpolatedElement::PRefine(int porder)*.

   (d) Executar o método *void TPZCompMesh::ExpandSolution()*, de modo a compatibilizar o solução interpolada inicialmente com o novo número de graus de liberdade devido ao refinamento p.

As operações descritas acima são ilustradas no diagrama de seqüência da Figura (6.12).

*void MeshError(TPZCompMesh *fine, TPZVec<REAL> &ervec, void(*f)(TPZVec<REAL> &loc, TPZVec<REAL> &val, TPZFMatrix &deriv),TPZVec<REAL> &truervec)*

Figura 6.12: UniformlyRefineMesh



Este método já estava implementado para o cálculo através do método global, tendo sido aqui adaptado, de modo a calcular o erro apenas nos elementos relativos ao elemento de referência do *patch* e em seus filhos.

Os parâmetros passados para o método são os seguintes:

- *fine*: malha computacional uniformemente refinada em *hp*. Esta malha é obtida através da utilização do método *void TPZCloneCompMesh::UniformlyRefineMesh()*, descrito anteriormente.
- *ervec*: deve ser passado o vetor de erros relativo à malha de referência, ou seja o número de elementos e a indexação deste vetor devem corresponder ao vetor de elementos computacionais da malha original.
- f: solução analítica para o problema, caso esta seja conhecida.
- *truervec*: será retornado o vetor de erro real, caso tenha sido fornecida a solução analítica. As considerações feitas para *ervec* também são válidas para este caso.

As operações realizadas, conforme descrito para o método global e aqui adaptadas, são as seguintes:

1. Criar um objeto análise e resolver a malha fina. Nesta implementação específica foi utilizado uma análise do tipo *Skyline*.
2. Obter a referência para a malha clone geométrica. Esta malha será utilizada para se obter as referências para a malha a ser adaptada.
3. Para cada elemento computacional componente da malha fina:
   (a) Verifica-se se o elemento é proveniente de um espaço interpolado (*TPZInterpolatedElement*);
   (b) Identifica-se o elemento geométrico associado ao elemento computacional em análise;
   (c) Verifica-se se o elemento geométrico identificado é o elemento de referência da malha clone geométrica ou um filho deste. Isto é feito através da função *int TPZCompCloneMesh::IsFather(TPZGeoEl *gel)*. Em caso negativo passa-se para próximo elemento computacional. em caso positivo continua-se a análise.
   (d) A definição do nível de refinamento é feita através de procedimento recursivo de obtenção do elemento pai do elemento da malha fina, até que este seja igual ao elemento da malha clone associado, fato que ocorre pelo refinamento ser hierárquico. Tendo-se o elemento pai do elemento da malha fina, correspondente ao elemento da malha clone, é possível a identificação do elemento geométrico correspondente

a este e assim, como descrito anteriormente, para a análise global, calcular a transformação entre os elementos mestre associados, possibilitando a transformação dos espaços necessária ao cálculo da diferença da solução, a qual é realizada na função *REAL TPZCompCloneMesh::ElementError(TPZInterpolatedElement *fine, TPZInterpolatedElement *coarse, TPZTransform &tr, void (*f)(TPZVec<REAL> &loc, TPZVec<REAL> &val, TPZFMatrix &deriv),REAL &truerror)*, sendo esta função inteiramente aproveitada do código global.

(e) Tendo o erro calculado para o elemento, o passo seguinte é identificar a qual elemento da malha original corresponde o erro calculado. Isto é feito através da função *int TPZCompCloneMesh::GetOriginalElementIndex(int clonelindex)*, a qual será descrita posteriormente.

(f) O erro calculado é então somado ao elemento correspondente no vetor de erros. Caso tenha sido fornecida a função analítica o erro real será adicionado ao elemento correspondente do vetor de erros real.

4. Tendo se percorridos todos os elementos da malha fina teremos calculados os erros de todos os elementos relacionados ao elemento de referência do *patch* ao qual a malha clone referencia.

A implementação do método é ilustrada pelo diagrama de seqüência apresentado na Figura (6.13).

*REAL ElementError (TPZInterpolatedElement *fine, TPZInterpolatedElement *coarse, TPZTransform &tr, void (*f)(TPZVec<REAL> &loc, TPZVec<REAL> &val, TPZFMatrix &deriv),REAL &truerror)*

Conforme mencionado anteriormente, este método foi inteiramente aproveitado da implementação do método global, não sendo necessário nenhuma compatibilização para sua inserção nesta classe.

A descrição do método pode ser encontrada no ítem relativo à implementação do método global.

int GetOriginalElementIndex(int elindex)

Esta função retorna o índice do elemento associado ao elemento dado como argumento, na malha de referência. Caso o elemento fornecido, através do seu índice no vetor de elementos da malha clone, não tenha elemento associado na malha de referência, a função retornará (-1).

As operações necessárias a obtenção deste índice são as seguintes:

1. Obter a malha clone geométrica associada à malha clone corrente. Isto é feito através da utilização de um *cast* para a malha clone geométrica na função *TPZGeoMesh * TPZCompMesh::Reference()*. Isto pode ser feito pelo fato de obrigatoriamente esta malha ter como malha geométrica associada uma malha do tipo clone geométrico.

Figura 6.13: MeshError

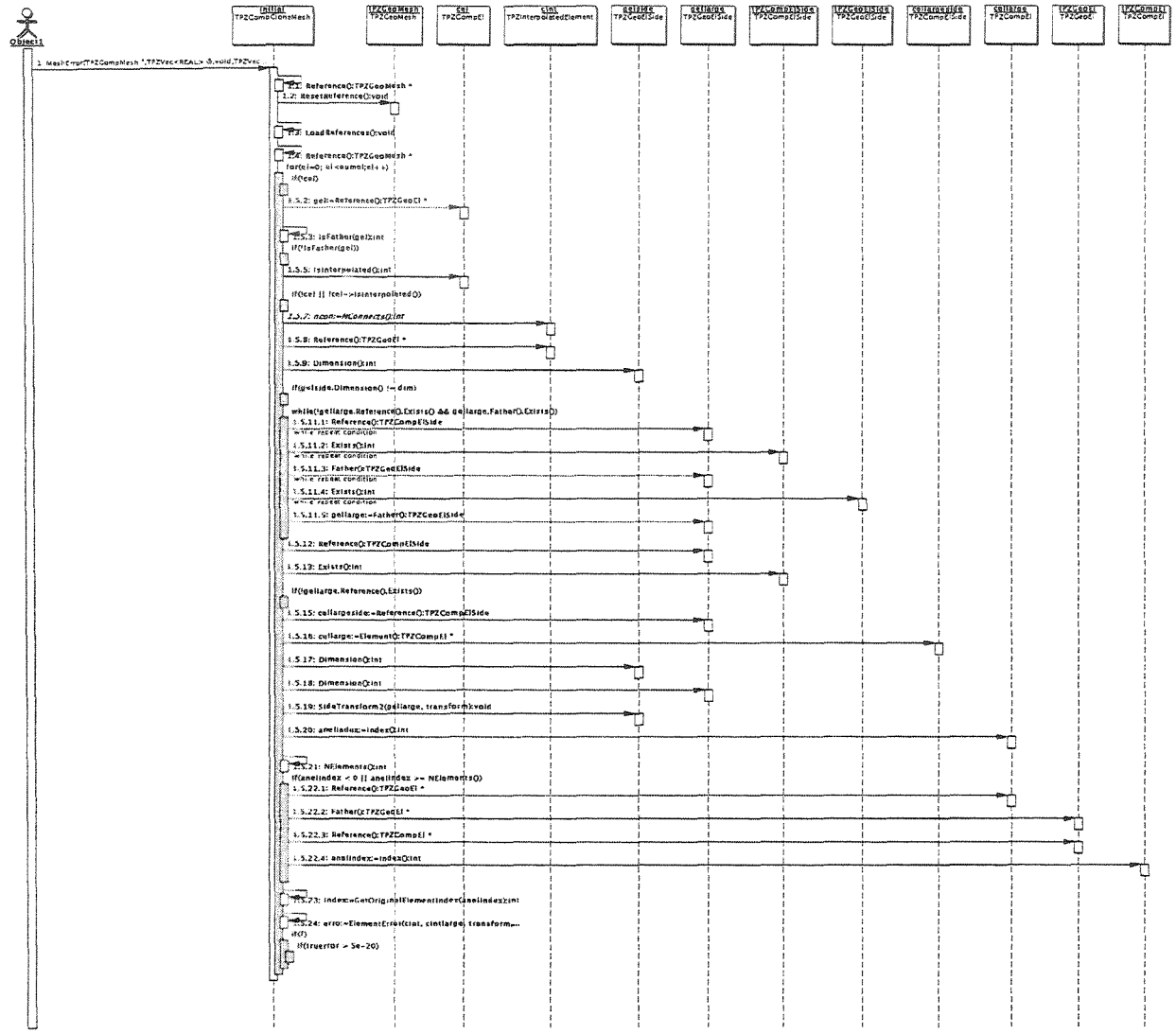
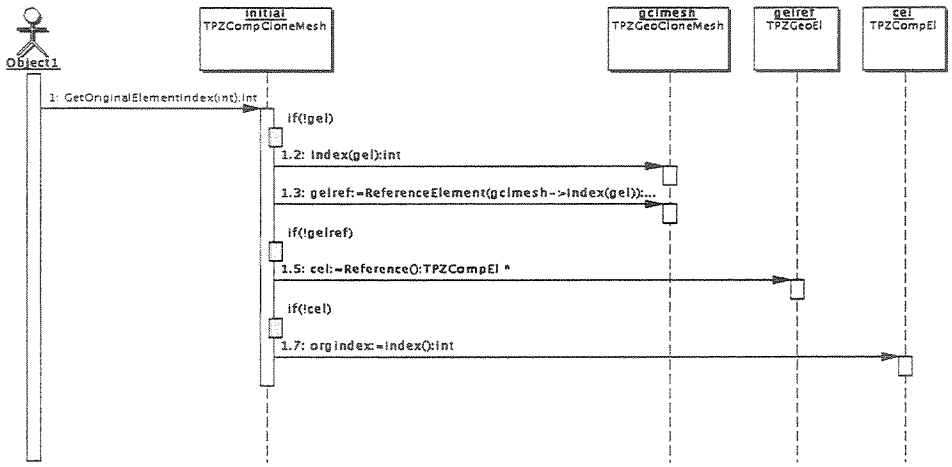


Figura 6.14: *GetOriginalElementIndex*



2. O passo seguinte é obter o elemento geométrico associado ao elemento do qual se deseja obter o elemento de referência associado. Isto é feito através da função *TPZGeoEl * ::TPZCompEl Reference()*, aplicada ao elemento computacional analisado.

3. Tendo o elemento geométrico da malha clone, utilizamos a função *TPZGeoEl * TPZGeoCloneMesh::ReferenceElement(TPZGeoEl * clonegel)*, para obter o elemento geométrico associado na malha geométrica de referência associada.

4. Com o elemento geométrico da malha original, tendo que a malha geométrica referencia a malha computacional de referência, basta aplicar a função *TPZCompEl * TPZGeoEl::Reference()* sob o elemento geométrico da malha original para obter um ponteiro para o elemento computacional associado.

5. Sobre o elemento computacional obtido no passo anterior é aplicada a função *int TPZCompEl::Index()*, a qual retorna o índice do elemento dado na malha a que ele pertence, no caso a malha computacional de referência.

6. Em caso de falha em qualquer ponto do método, é retornado -1, para indicar que não há elemento computacional associado ao elemento dado.

O diagrama de seqüência da Figura (6.14) mostra a implementação deste método.

*TPZInterpolatedElement *TPZCompCloneMesh::GetOriginalElement(TPZCompEl *el)*

Este método utiliza o método anterior para a obtenção do elemento associado, acrescentando ao anterior o fato de utilizar o índice devolvido por aquele para a obtenção do ponteiro

para o elemento. Isso é feito simplesmente acessando o vetor de elementos da malha original, ou seja: *fCloneReference->ElementVec()[orgindex].*

*void ApplyRefPattern(REAL minerror, TPZVec<REAL> &error, TPZCompMesh *fine, TPZStack<TPZGeoEl *> &gelstack, TPZStack<int> &porder)*

Este método é o método que gerencia o processo de adaptação dos elementos dentro desta classe.

Os parâmetros passados para o método são os seguintes:

- *minerror*: é fornecido um erro de corte, erro mínimo que o elemento deve apresentar para mostrar necessária a análise de seu padrão de refinamento;
- *error*: corresponde ao vetor de erros, relativo aos elementos da malha original, calculado anteriormente através do método *void TPZCompCloneMesh::MeshError(...)*;
- *fine*: corresponde à malha clone uniformemente refinada, obtida anteriormente através da função *TPZCompMesh* TPZCompCloneMesh::UniformlyRefineMesh()*;
- *gelstack*: esta lista de elementos contém a lista de elementos geométricos refinados em h. Caso a posição esteja nula indicará a existência de refinamento p puro para o elemento;
- *porder*: contém a lista de ordens de refinamento p para cada elemento correspondente de *gelstask*;

As operações aqui realizadas são as seguintes:

1. Obtém-se a malha clone geométrica associada à malha clone, sendo isto feito através da função *TPZGeoMesh* TPZCompMesh::Reference()*, sendo feito o *cast* para a malha clone geométrica, conforme já descrito.
2. Retiram-se as referências da malha clone geométrica da malha clone e as carregam na malha fina, ou seja *geoclonemesh->ResetReference()* e *fine->LoadReferences()*.
3. Para todos os elementos da malha clone refinada:
   (a) Verificar se o elemento faz parte do elemento de referência do *patch*. Isto é feito através da função *int TPZCompCloneMesh::IsFather(TPZGeoEl*)*. Em caso negativo passa-se para o próximo elemento e em caso afirmativo continua-se a análise;
   (b) Obtém-se o índice do elemento computacional associado ao elemento analisado. Para tal é utilizada a função *int TPZCompCloneMesh::GetOriginalElementIndex(int cloneindex)*.

(c) Tendo-se o índice do elemento na malha original verifica-se qual o erro associado a este elemento no vetor de erros *error*. Caso o erro associado ao elemento seja menor que o erro mínimo de análise insere-se o próprio elemento geométrico na lista de gelstack e a ordem é definida como sendo a mesma do elemento original, passando-se então para a analise do próximo elemento. Em caso contrário continua-se a análise.

(d) Aplica-se o método void TPZCompCloneMesh::AnalyseElement(TPZOneDRef &f, TPZInterpolatedElement *cint, TPZStack<TPZGeoEl *> & gelstack, TPZStack <int> &porder). Esta função já estava implementada para o método global, já estando descrita.

O diagrama de seqüência da Figura (6.15) ilustra o método.

*void AnalyseElement (TPZOneDRef &f, TPZInterpolatedElement *cint, TPZStack<TPZGeo. *> &subels, TPZStack<int> &porders)*

Este método também foi inteiramente aproveitado da implementação do método global, estando sua descrição já realizada na seção relativa à implementação do método global. Apenas uma adaptação foi feita ao método que é a inserção dos parâmetros de refinamento *hp* na posição relativa ao elemento de referência do elemento analisado, ou seja, obtém-se o índice do elemento da malha original através da função *int TPZCompCloneMesh::GetOriginalElementIndex(int cloneindex)* , já descrita.

void TPZCompCloneMesh::DeduceRefPattern(TPZVec<TPZRefPattern> &refpat, TPZVec<int> &cornerids, TPZVec<int> &porders, int originalp)

Utilizou-se a mesma implementação realizada para o modelo global. A descrição deste método pode ser encontrada na seção relativa a esta implementação.

*int IsFather(TPZGeoEl *el)*

Esta função verifica se um dado elemento é o elemento de referência do *patch* ou um de seus filhos, caso em que a função retornará 1. Em caso contrário a função retornará 0.

O argumento da função é o elemento geométrico, relativo à malha clone.

As operações realizadas pelo método são as seguintes:

1. Verifica se o elemento informado corresponde ao elemento de referência do *patch*. Em caso afirmativo retorna 1 e sai da função.

2. Caso contrário substitui-se o elemento dado pelo seu pai - função *TPZGeoEl * TPZGeoEl::Father()*, caso este exista, e verifica-se se este elemento corresponde ao elemento de referência do *patch*.

3. O passo 2 é repetido até que se encontre um ancestral do elemento que corresponda ao elemento de referência, ou que se obtenha pai nulo, o que implicará que todos os ancestrais do elemento foram analisados e nenhum deles corresponde ao elemento de referência do *patch*.

Figura 6.15: *ApplyRefPattern*

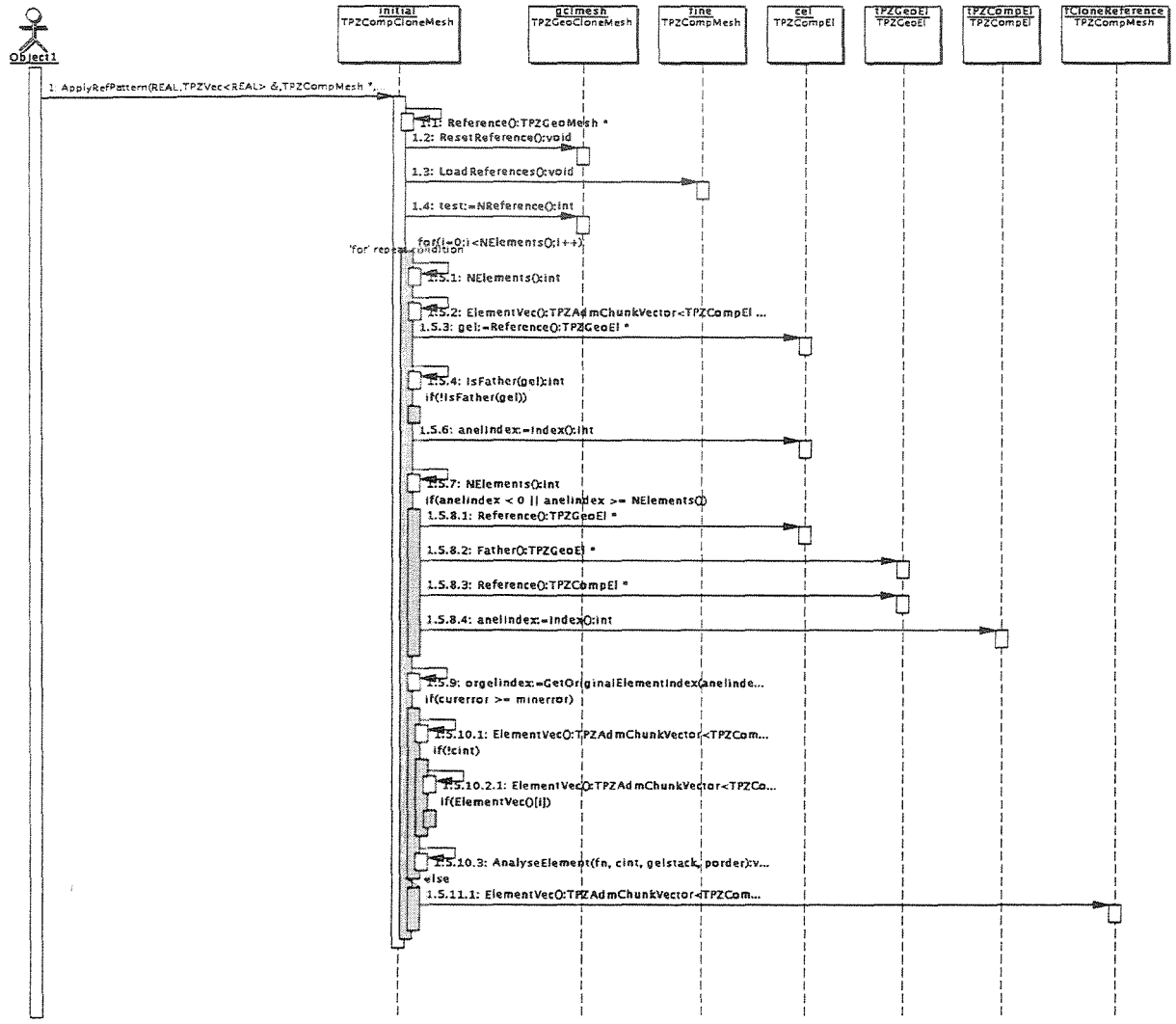
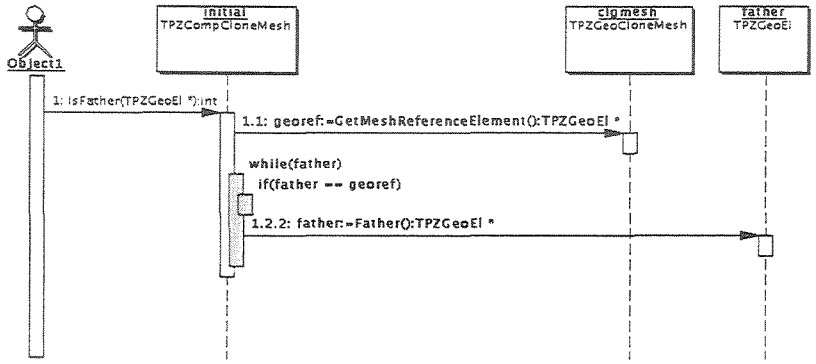


Figura 6.16: *IsFather*



A implementação desse método é mostrada na Figura (6.16).

void Print(ostream &out)

Imprime as variáveis da classe.

Capítulo 7

Resultados Obtidos - Testes de Validação

Os testes de implementação do código gerado consistiram da utilização de malhas simples, tais como um quadrilátero uniformemente refinado em h. Com estas malhas foi possível depurar os resultados intermediários.

Os testes de validação realizados são descritos na seqüência.

7.1 Problema de Laplace para Placa em L

Como primeiro teste de validação foi utilizado o mesmo modelo apresentado na Figura (5.11) e também utilizado em [6]. Os elementos utilizados também foram quadrilaterais e as condições de contorno impostas são iguais ao gradiente da solução analítica nos bordos, da mesma forma que para o modelo implementado na validação do modelo global.

Este modelo, utilizando a técnica local aqui implementada, tem seus resultados apresentados na seqüência, sendo na Figura (7.1) apresentado na seqüência o erro estimado junto a ao erro real, a efetividade do estimador de erros e o tempo de processamento. Nestes gráficos é possível visualizar a convergência do erro estimado, a sua efetividade, ou seja, com qual qualidade que este aproxima o erro real um parâmetro muito importante e que viabiliza a utilização do método que é o custo computacional, medido através do tempo de processamento.

Na Figura (7.2) é mostrada a malha obtida em 2, 5 e 10 passos adaptativos, enquanto na Figura (7.3) é mostrado o detalhe do L para 10 passos.

Conforme pode ser verificado na Figura (7.1), o estimador de erros converge com taxa similar à taxa de convergência do erro real, sendo o padrão de convergência quadrático, da mesma forma que o valor obtido pelo método de estimação global. A efetividade apresentada também mostrou-se adequada, oscilando entre 80% e 88%, não existindo tal oscilação no modelo global, entretanto a efetividade deste tende a 80%. Já o tempo de processamento mostrou-se muito menor que o tempo de processamento para o método global.

Isto já era de se esperar, uma vez que o número de equações de todas sub-malhas é

aproximadamente igual, dependendo de suas vizinhanças e da ordem p de seus elementos.

Já para o processo global este aumento de tempo mostra-se exponencial, uma vez que a cada refinamento o número de equações é aumentado pelo quadrado do número de graus de liberdade acrescidos.

A Figura (7.2) demonstra, qualitativamente, que o processo de refinamento concentra-se na região de singularidade do problema, sendo mostrado na Figura (7.3) a concentração de elementos de alta ordem p na região do L.

7.2 Comparação: Modelo Global vs. Modelo Local

Para facilitar esta comparação, foi feito um exemplo no qual os modelos global e local são utilizados para adaptar a mesma malha.

O procedimento consiste em se tendo uma malha, obter a malha através do método de estimação de erro local, sendo armazenados os dados de erro efetividade e tempo para esta análise. Em seguida obtém-se a malha adaptada pelo método global, sendo também armazenados os dados de erro, efetividade e tempo.

Os resultados desta comparação são mostrados nos gráficos da Figura (7.4).

Observe que ambos os modelos conduziram a índices de efetividade acima de 80%, podendo ser verificado como grande vantagem do modelo local o fato de seu tempo de obtenção das malhas adaptadas ser muito menor que o tempo obtido pelo modelo global.

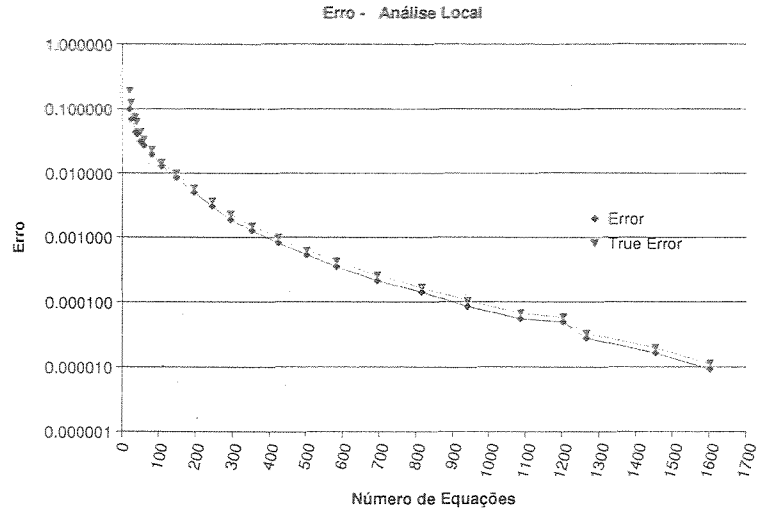
A curva de tempo do modelo local apresentou-se com o perfil de um polinômio quadrático, com gradiente baixo, podendo ser em alguns trechos aproximada por uma reta. Este perfil pode ser explicado pelo fato de um *patch*, de acordo com a estratégia adotada, ter sempre um número pequeno de elementos em relação a malha global, sendo o crescimento do tempo de obtenção da malha pelo método local decorrente do aumento do refinamento p dos elementos e pelo aumento do número de *patches*. Isto justificaria o aumento praticamente linear do tempo, para o número de equações testadas.[1]

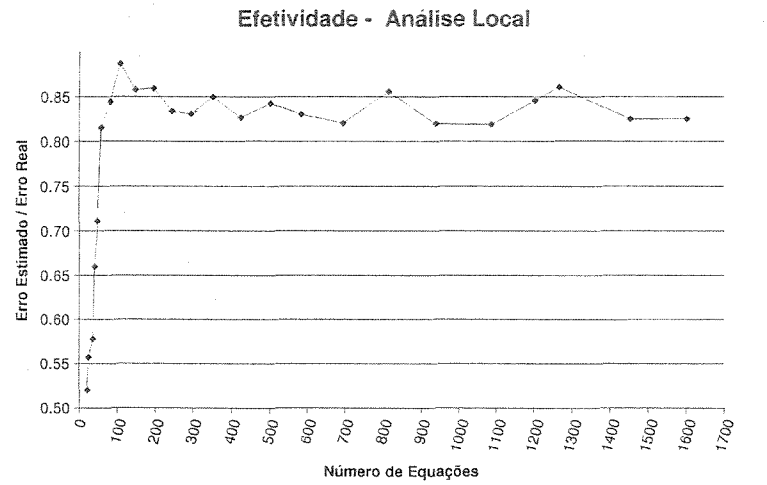
Já o modelo global tem seu gráfico de tempo com perfil exponencial, sendo isto facilmente justificável pelo número de equações aumentar pelo quadrado do número de graus de liberdade acrescido a malha.

Com relação aos dados de efetividade, observa-se uma grande oscilação do índice de efetividade nos resultados para o modelo local. Conjectura-se que esta oscilação deve-se ao fato da malha utilizada como malha de referência em cada passo ser a malha adaptada pelo método global no passo anterior. De certo modo o padrão de refinamento indicado pelo método global pode diferir daquele identificado pelo método local. O gráfico de efetividade calculado sobre malhas obtidas com o método local mostrou-se muito mais comportado. (ver Figura (7.1)).

[1] Isto é válido após alguns passos de refinamento, pois nos passos iniciais, cada patch pode ter até o número de elementos total da malha. Neste caso é lógico que este terá um tempo de obtenção da malha adaptada maior.

Figura 7.1: Resultados para a malha quadrilateral





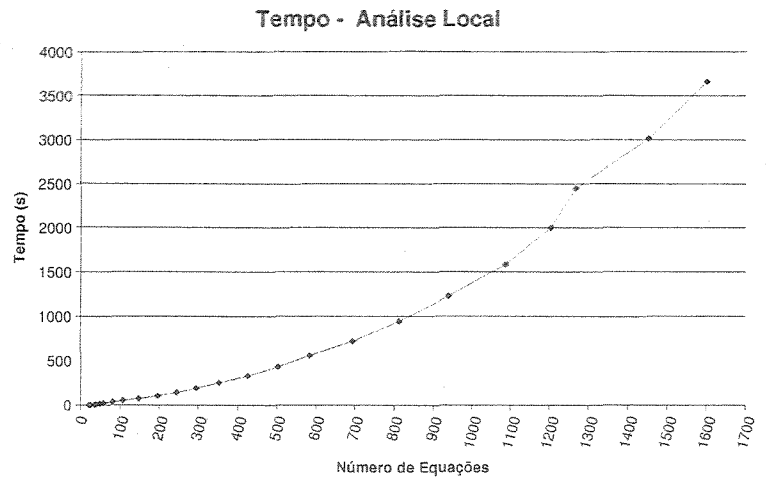


Figura 7.2: Malhas obtidas (2, 5 e 10 passos adaptativos) - Escala de cores indica a ordem de refinamento p

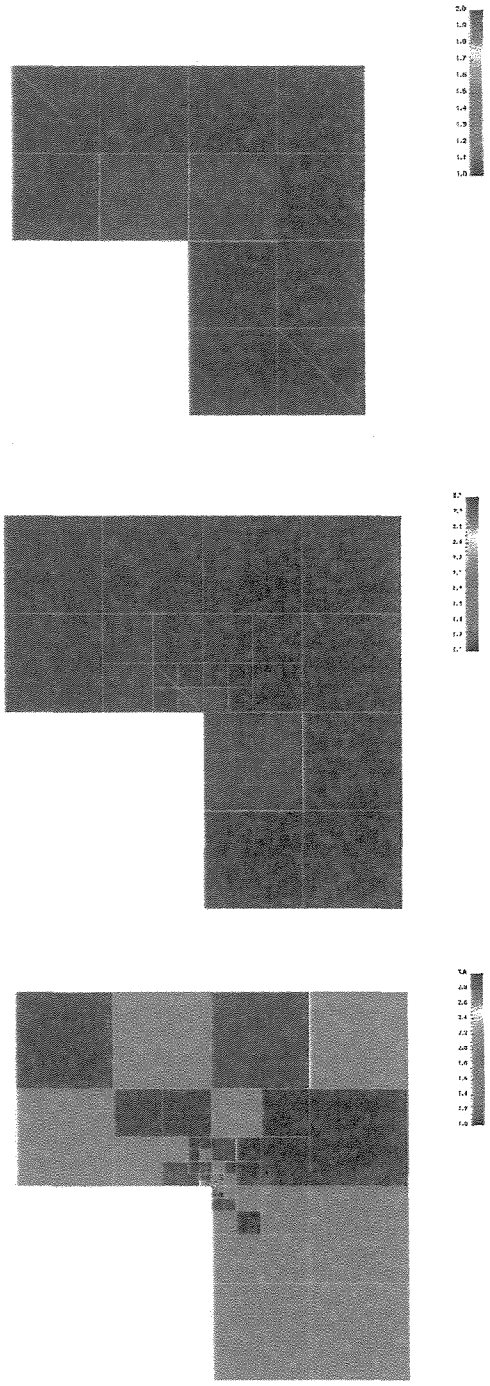


Figura 7.3: Detalhe da malha adaptada (passo adaptativo = 10) - Escala de cores indica ordem de refinamento p

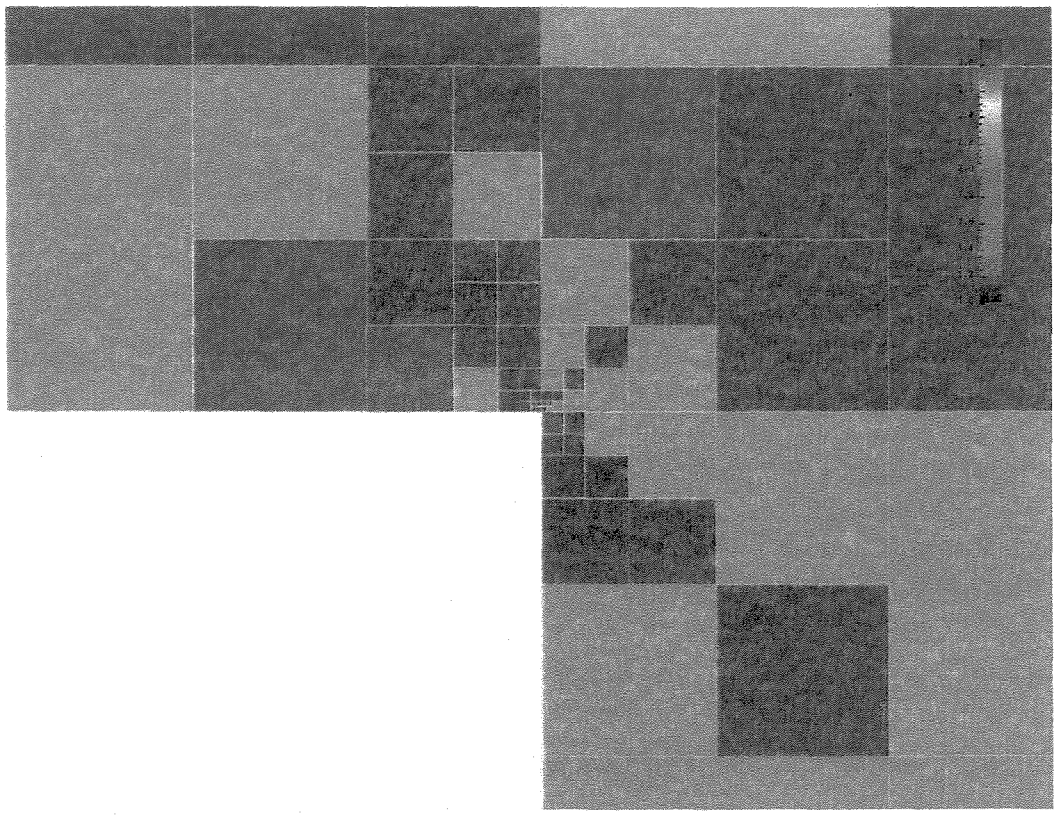


Figura 7.4: Comparação dos Modelo Global e Local

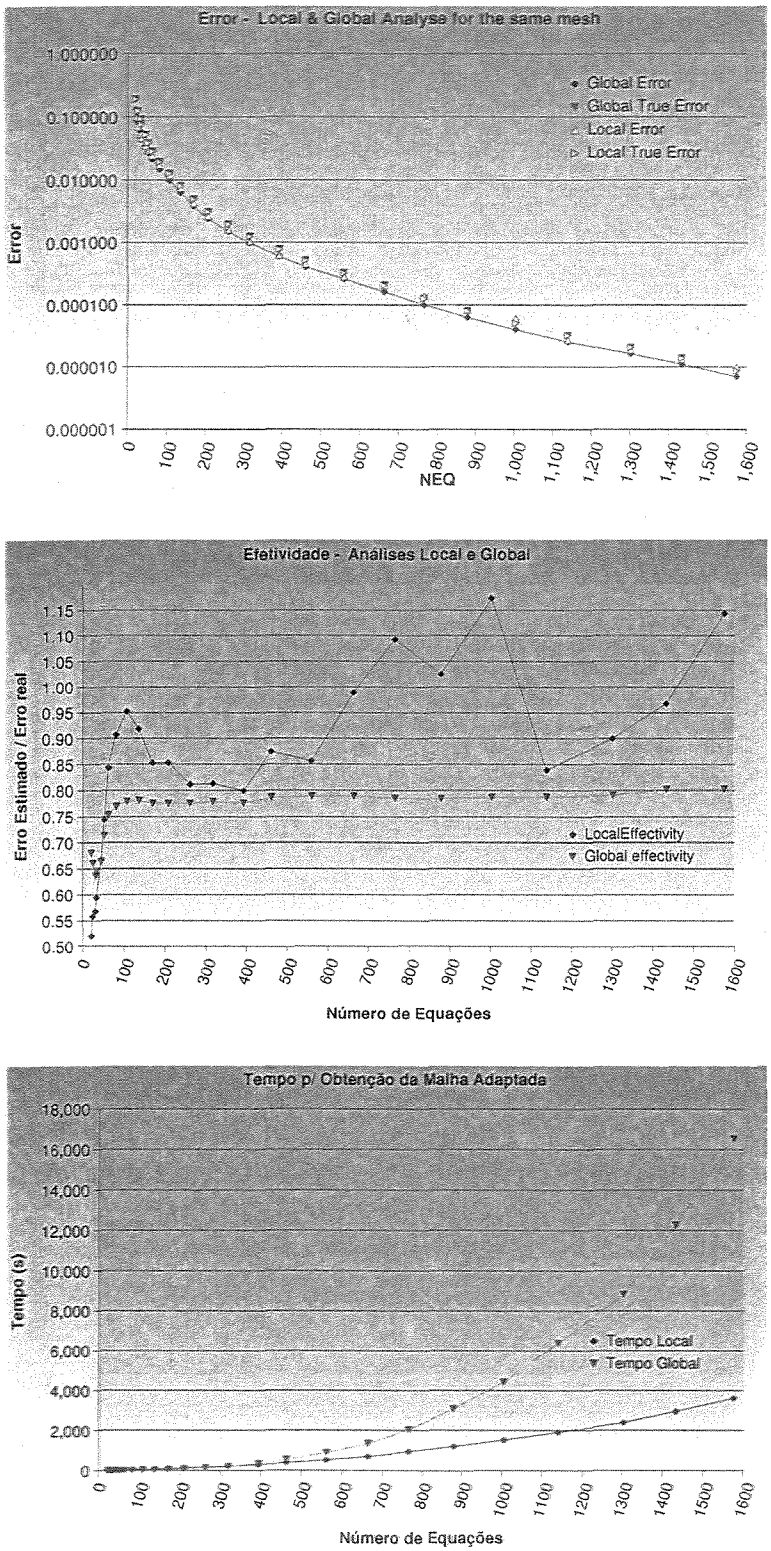
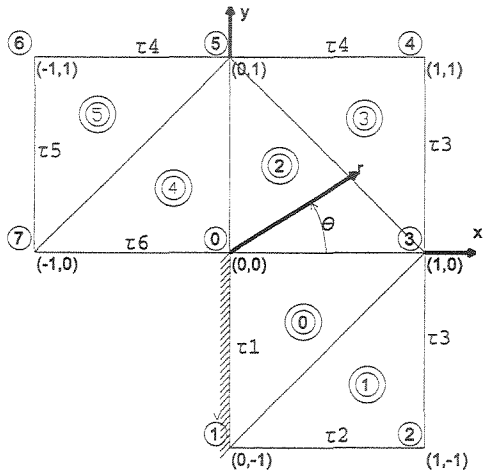


Figura 7.5: Malha de elementos triangulares utilizada



7.3 Verificação da Utilização de Elementos Triangulares

Para verificar a confiabilidade do método, o mesmo modelo utilizado acima foi analisado com a utilização de elementos Triangulares. A malha utilizada é mostrada na Figura (7.5).

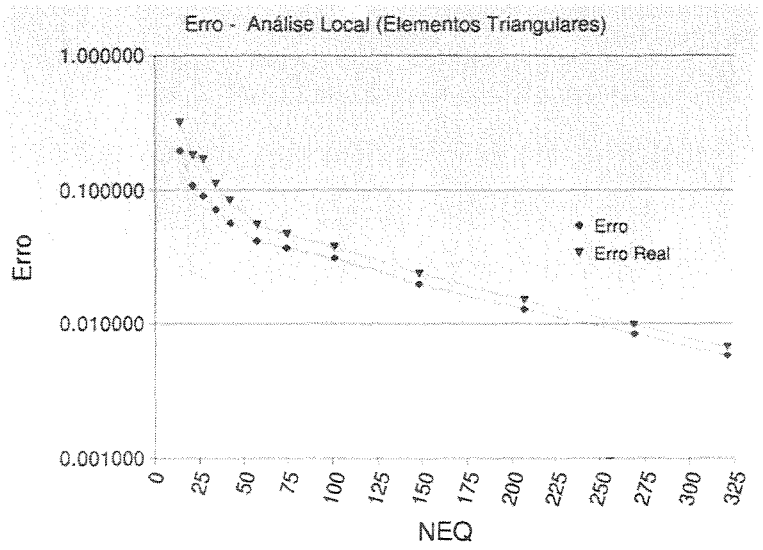
Os resultados são mostrados na Figura (7.6), sendo destacado o fato do número de equações alcançadas ser menor pelo fato do refinamento no canto do L ter levado alguns elementos a ter uma deformação tal, a ponto de causar instabilidade numérica durante a resolução do problema. A Figura (7.7) mostra a malha após 2, 5 e 10 passos adaptativos respectivamente. A Figura (7.8) mostra um detalhe da região do L.

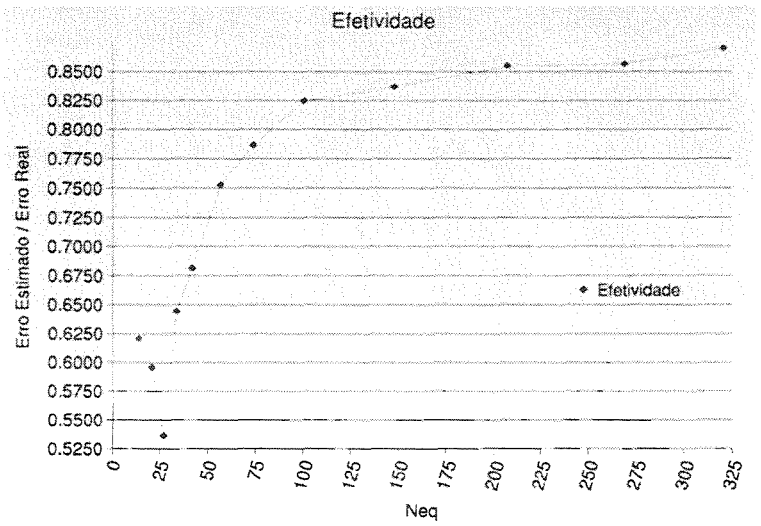
Conforme pode ser verificado, o padrão de convergência é similar ao obtido para os elementos quadráticos (para o mesmo número de equações), sendo a efetividade próxima a 85% também para este caso. O tempo de processamento foi compatível com os resultados obtidos para a malha de elementos quadrilaterais.

Com relação às malhas adaptadas, pode-se também verificar a concentração de elementos na região do ponto de descontinuidade que é o canto do L.

Conforme pode ser observado nos resultados, o método auto-adaptativo conduziu à malhas com erro decrescente, entretanto, dado o número máximo de equações não ter sido razoável, algo próximo a 1000 equações, não há meios de concluir algo sobre as taxas de convergência e de efetividade.

Figura 7.6: Problema de Laplace - Modelo com Elementos Triangulares





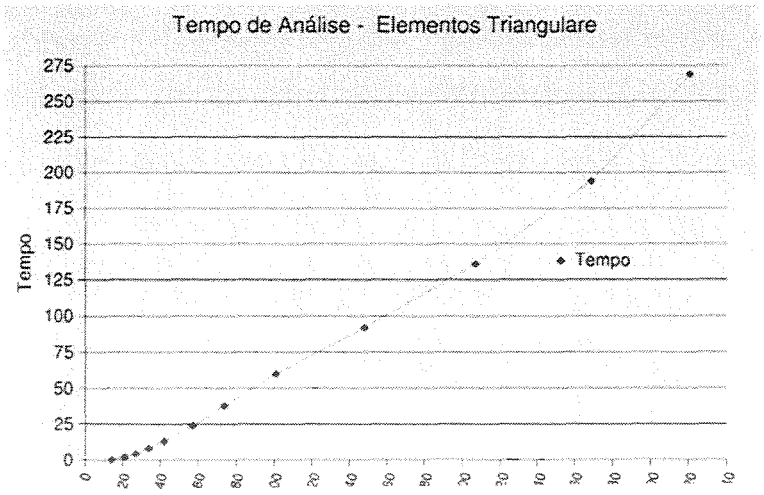


Figura 7.7: Malha Triangular Adaptada (2, 5 e 10 passos adaptativos) - Escala de core indica a ordem de refinamento p

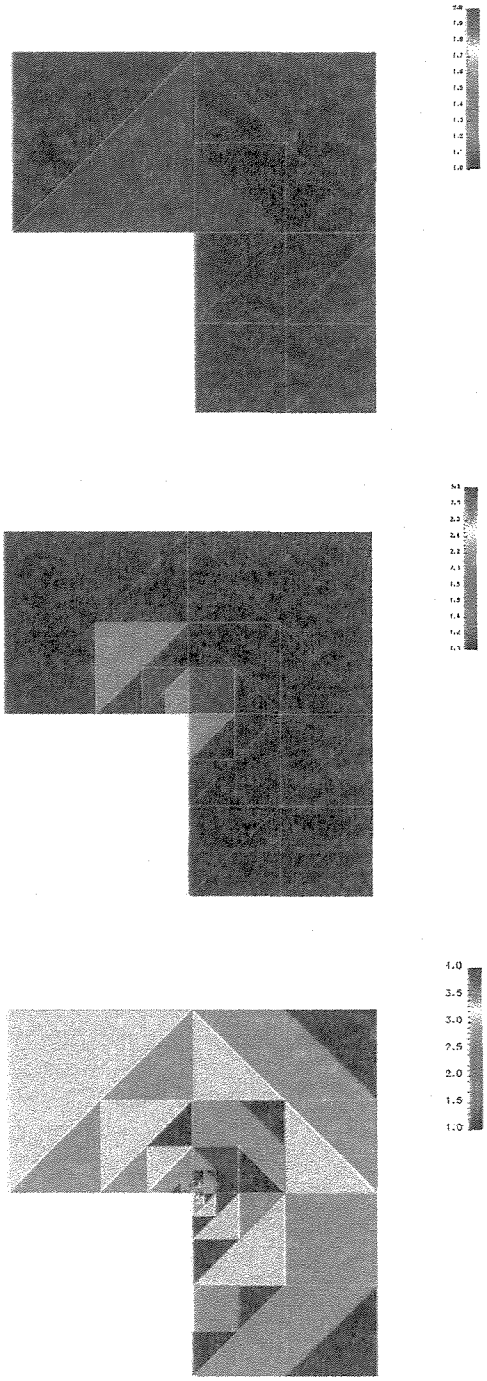
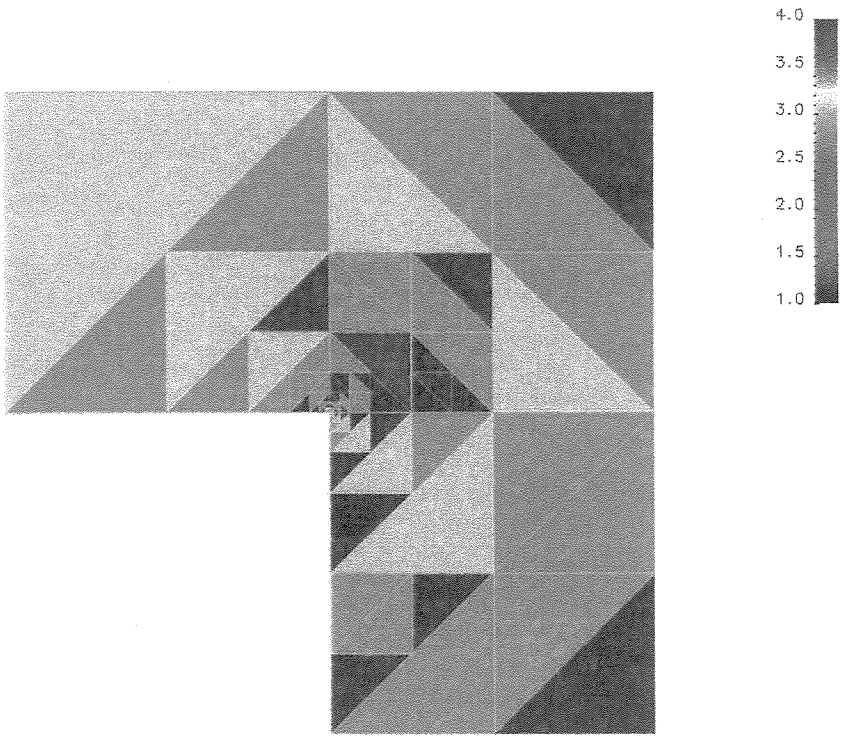


Figura 7.8: Detalhe da região do L (passo adaptativo = 10) - Escala de cores indica a ordem de refinamento p



Capítulo 8

Conclusão

Implementou-se um método auto-adaptativo *hp* baseado em uma estimativa de erro local. A técnica foi testada e validada para modelos bi-dimensionais. Modelos tri-dimensionais foram testados, mas não validados.

O erro estimado localmente, através do modelo implementado, teve um índice de efetividade similar ao erro estimado por um modelo local, tendo como vantagem o tempo de processamento, que no caso do modelo analisado, enquanto o modelo global tem uma curva de tempo exponencial, o modelo global tem um perfil próximo ao linear.

Ainda como detalhe adicional, caso a análise local seja paralelizada, este aumento de tempo será muito menor em relação ao obtido até agora.

Estes resultados promissores mostram que esta técnica ainda pode ser muito explorada, principalmente nos aspectos relativos a análise de desempenho, paralelização de tarefas e estudo de implementação de outros tipos de estimadores de erro utilizando esta abordagem local.

Com relação à extensão deste método para problemas tri-dimensionais, durante os testes com o método, foram utilizadas funções cuja validação não havia sido feita anteriormente.

Desta forma, verificou-se uma inconsistência de dados em alguns padrões de refinamento específicos. Esta inconsistência está sendo analisada e tão logo esteja validado o trecho de código utilizado, voltar-se-á a validação do método para problemas tri-dimensionais.

Acredita-se que com este código validado o funcionamento do código auto-adaptativo será imediato, uma vez que para o modelo tri-dimensional utilizado, chegava-se a sete passos de refinamento, com a estrutura das malha clone corretas. Neste passo era realizado um padrão de refinamento *hp* cuja implementação apresentou problema.

De modo a validar o modelo tri-dimensional será utilizado o problema modelo adotado em [7].

Ressalta-se que a estrutura do método está implementada e mostra-se promissora em termos de utilização prática, pois a redução do custo computacional para a obtenção de uma malha adaptada foi sensível.

8.1 Extensões

Conforme já mencionado, as principais extensões do trabalho estão ligadas à melhora de seu desempenho, haja vista que este aspecto não foi analisado com muita profundidade no presente trabalho.

Outra extensão natural é a paralelização do código, sendo um código destes de extrema utilidade para resolução de grandes problemas de engenharia, e que requerem adaptatividade, tais como problemas de mecânica dos fluidos computacional, onde a análise de problemas ao longo do tempo requerem um grande esforço computacional para a obtenção de malhas adequadas.

Um aspecto não destacado anteriormente, más que também pode ser estudado é a obtenção de um padrão que conduza ao erro menor que o admissível diretamente pela análise local, isso é dito pelo modelo atual funcionar da seguinte forma: dada uma malha com solução conhecida, isolam-se *patches* de elementos, refinam-se estes *patches* uniformemente em *hp* e faz se a análise do padrão de refinamento dos elementos com erro considerável. Os elementos analisados tem o seu padrão de refinamento estudado e o padrão adotado é aquele com um número de equações menor que o número de equações dado pelo refinamento uniforme e cujo erro é mínimo para este número de equações.

Observe que este padrão é adotado independentemente do erro. Assim, algo a ser analisado, através do padrão de refinamento obtido, caso o erro ainda seja considerável, realizar um novo passo de refinamento neste *patch* localmente, antes de transferir o padrão de solução, sendo isto feito de maneira recursiva até que o erro obtido para os elementos componentes do *patch* seja menor que um limite aceitável.

De fato isto pode levar a um refinamento excessivo em alguns casos más, por outro lado, pode reduzir sensivelmente o número de passos de adaptatividade e ainda reduzir consideravelmente o tempo de análise.

Anexos

Apêndice A

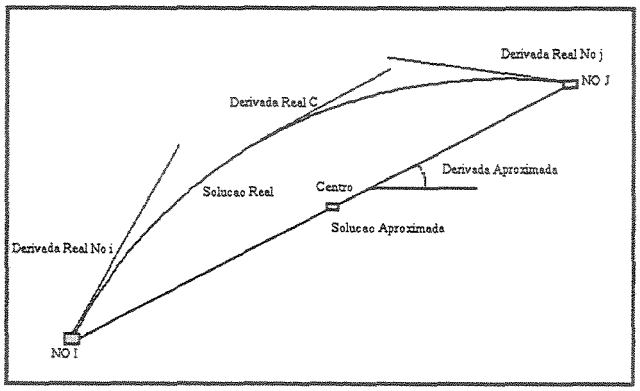
Estimadores Baseados em Valores Suavizados - Estimador de Erros de Zienkiewickz-Zhu (*Patch Recovery Technique*)

No cálculo de elementos finitos, o resultado da análise é a obtenção da função que representa a variável de estado. Entretanto, normalmente, também procura-se pela distribuição das funções de fluxo ao longo do domínio. Assim, no caso da elasticidade, os engenheiros de estruturas estão preocupados em obter as tensões atuantes no corpo, pois partindo-se destas é possível realizar o dimensionamento.

Dessa forma, existe a necessidade de realizar o pós-processamento dos valores obtidos para encontrar o gradiente da função analisada. Aqui surge um novo porém, normalmente a função que representa o gradiente é descontínua entre as bordas dos elementos, tornando-se necessário para sua representação, suavizá-la, de modo a torná-la contínua ao longo do domínio.

A suavização pode ser feita de diversas formas, entretanto, foi demonstrado por Zlamal [27, 28] que no centróide do elemento, a função gradiente apresenta a propriedade de super-convergência, ou seja, neste ponto a função gradiente convergirá mais rapidamente para seu valor real. Tal fato pode ser visualizado, em uma dimensão, na Figura (A.1), onde é mostrado que para a função variável de estado, os nós dos elementos apresentam a propriedade de super-convergência, enquanto sua aproximação para a função gradiente é pobre, fato contrário ao que ocorre no centro do elemento.

Figura A.1: Super-convergência do Gradiente



A.1 Processo de Cálculo

Para mostrar o processo de cálculo do erro através desse método, tomemos o seguinte problema modelo:

$-\nabla u + cu = f$,

com as seguintes condições de contorno:

$\frac{\partial u}{\partial n} = g$ em Γ_N e $u = 0$ em Γ_D

O erro, em termos da norma de energia, será dado por:

$\|e\|^2 = \int_\Omega |\nabla u - \nabla u_X|^2 \, dx$, sendo:

∇u: Gradiente da função u real;

∇u_X: Gradiente da função u_X calculada numericamente no elemento K;

Como, em geral, não temos o valor da função gradiente real, substituiremos o seu valor pelo valor da função suavizada que, caso seja adequadamente escolhida, conforme será descrito adiante, representará uma aproximação melhor do que aquela calculada isoladamente no elemento.

Dessa forma teremos que o erro estimado, associado ao elemento K será:

$\eta_K^2 = \left\| G_X(u_X) - u_X^l \right\|^2 dx$, sendo:

$G_X(u_X)$: Gradiente da função u_X suavizada;

u_X^l: Primeira derivada da função u_X calculada numericamente no elemento K;

Como há a necessidade do pós processamento da solução, esse processo agrega um custo computacional muito pequeno ao processo, fornecendo bons resultados.

O processo pode ser resumido então nas seguintes etapas:

- executar o pós-processamento dos resultados de modo a obter o gradiente da solução em cada elemento;
- suavizar a função gradiente ao longo do domínio;

- calcular a norma do erro através do somatório, em todos os elementos, da diferença entre o valor suavizado e o valor inicial.

O diferencial que pode ser agregado nesse processo é a forma de suavizar a função gradiente.

Como pode ser visto em [1], os operadores de suavização devem, essencialmente respeitar as seguintes condições:

- Condição de Consistência: a função utilizada para suavização deve conduzir aos valores reais da função, quando as condições forem favoráveis (malha de dimensão adequada). Como exemplo, caso utilizemos a média entre os valores dos gradientes calculados nos nós de um elemento de barra, quanto mais reduzirmos o tamanho do elemento, mais próximo ficará o valor suavizado do valor real;
- Condição de Obtenção: a obtenção do valor suavizado não deve implicar em custos computacionais adicionais;
- Facilidade de Manipulação: a função deve ser facilmente manipulável, possibilitando que as funções obtidas sejam facilmente integráveis.

A.2 Operador de Zienkiewicz-Zhu

O operador de suavização de Zienkiewicz-Zhu consiste nas seguintes etapas para o cálculo do valor suavizado em um determinado nó:

1. Selecionar o nó onde se quer determinar o valor suavizado;
2. Identificar quais elementos tem esse nó na sua lista de conectividades;
3. Para cada elemento identificado, calcular o valor do gradiente no seu centróide, como sendo a média dos valores nos seus nós;
4. Com o valor no centróide de todos os elementos, é ajustada uma função, por mínimos quadrados a todos esses nós.
5. Através da função ajustada é calculado o valor suavizado no nó selecionado em 1.

Assim, com o valor suavizado é possível obter-se a estimativa do erro, conforme descrito anteriormente.

Apêndice B

Implementação de Matriz Bloco Sobreposto

Como parte dos esforços para a generalização do ambiente PZ e como parte dos esforços para possibilitar a criação de códigos passíveis de paralelização, está inserida o aumento do número de classes algébricas, incluindo a implementação da classe para tratamento de uma matriz através de blocos que podem apresentar sobreposição.

A vantagem da utilização desse tipo de matriz é que além de agregar a possibilidade de acesso direto ao bloco elementar, proveniente da matriz bloco diagonal já implementada, essa classe, quando utilizada como pré-condicionador já agregará uma sobreposição da solução dos elementos em cada conectividade, o que esperamos ser um pré-condicionador com eficiência melhor do que a apresentada quando do uso de pré-condicionador baseado em bloco diagonal, cuja solução não prevê esta sobreposição.

B.1 Fundamento Algébrico

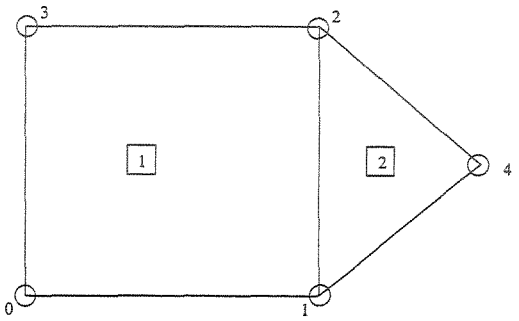
Dada uma matriz $[K]$, pode-se mostrar que essa pode ser obtida pelo somatório de suas sub matrizes, ou seja:

$$K = \Sigma_e K^e_{i,j} \tag{B.1}$$

Da mesma forma, com esta abordagem, o produto matriz vetor $K.u$, pode ser escrito como o somatório do produto dos sub blocos com o sub vetor correspondente, ou seja:

$$K.u = \Sigma_e K^e_{i,j}.u^e_j \tag{B.2}$$

Figura B.1: malha exemplo



B.1.1 Operação de Multiplicação

Dados os blocos componentes da matriz, incluindo seus índices de localização na matriz global e o vetor u global, a operação de multiplicação:

$$K.u = f$$

será realizada da seguinte forma:

Algorithm 2 Multiplicação através da utilização de blocos sobrepostos

1. Fornecer os blocos K_e componentes da matriz K, o vetor u e seus respectivos índices.
2. Decompor o vetor u em blocos u_e, correspondentes aos blocos dados (*Scatter*).
3. Calcular a multiplicação dos sub-blocos:

$$K_e.u_e = f_e$$

4. Com os sub vetores f_e é montado o vetor f (*Gather*)

Para o melhor entendimento da operação esta pode ser acompanhada através da seqüência abaixo, considerando a malha da Figura (B.1), onde é representada uma malha simples com dois elementos, sendo as conectividades do elemento $0 = \{0, 1, 2, 3\}$ e do elemento $1 = \{1, 4, 2\}$.

Abaixo é mostrado como seria feito, de maneira esquemática, as operações *Gather*, dado um vetor $u = \{u_0, u_1, u_2, u_3, u_4\}$ por conectividade e a operação *Scatter*, dado um vetor $f = \{f_0^0, f_1^0, f_2^0, f_3^0, f_0^1, f_1^1, f_2^1\}$ por bloco. Os índices superescritos indicam o elemento e os subescritos o número de seqüência do dado dentro do elemento.

$$\begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \Rightarrow Scatter \Rightarrow \begin{bmatrix} u_0^0 = u_0 \\ u_1^0 = u_1 \\ u_2^0 = u_2 \\ u_3^0 = u_3 \\ u_0^1 = u_1 \\ u_1^1 = u_4 \\ u_2^1 = u_2 \end{bmatrix} \begin{bmatrix} f_0^0 \\ f_1^0 \\ f_2^0 \\ f_3^0 \\ f_0^1 \\ f_1^1 \\ f_2^1 \end{bmatrix} \Rightarrow Gather \Rightarrow \begin{bmatrix} f_0 = f_0^0 \\ f_1 = f_1^0 + f_0^1 \\ f_2 = f_2^0 + f_2^1 \\ f_3 = f_3^0 \\ f_4 = f_1^1 \end{bmatrix}$$

B.2 Interface

Para a utilização desta classe, há a necessidade do conhecimento do número de blocos. Tendo este é possível a inicialização da matriz, onde os elementos de todos os blocos serão nulos. A Figura (B.2) mostra, de modo esquemático, a inicialização de um objeto desta classe.

No caso de inserção de dados, essa pode ser feita via o método *AddBlock*, ou por intermédio do método *AddKell* que em última instância utiliza também o método *AddBlock*. A Figura (B.3) mostra esquematicamente esta seqüência.

Já o cálculo da multiplicação, o qual utiliza os métodos *Scatter* e *Gather*, é mostrado de modo esquemático na Figura (B.4).

B.3 Implementação

A matriz bloco diagonal foi implementada através da derivação da classe *TPZBlockDiagonal*, que já apresenta todas as funcionalidades de armazenamento, acesso a dados e operações matriciais básicas.

Os membros desta classe são:

- *TPZBlockDiagonal fStoreMatrix*: objeto que armazena os blocos da matriz;
- *fBlockPos*: vetor com a posição na matriz do primeiro elemento do bloco;
- *fInitialized*: contém o número de blocos da matriz.

Construtores, Destrutores e Métodos de Inicialização

TPZOverlapBlock()

Cria um objeto com todos parâmetros nulos.

TPZOverlapBlock(TPZVec<int> blocksize, int rows)

Aloca espaço em memória para receber a quantidade de blocos componentes do vetor *blocksize*, sendo o tamanho de cada bloco indicado nos elementos componentes do vetor.

TPZOverlapBlock(TPZOverlapBlock &ovlbl)

Figura B.2: Inicialização de dados

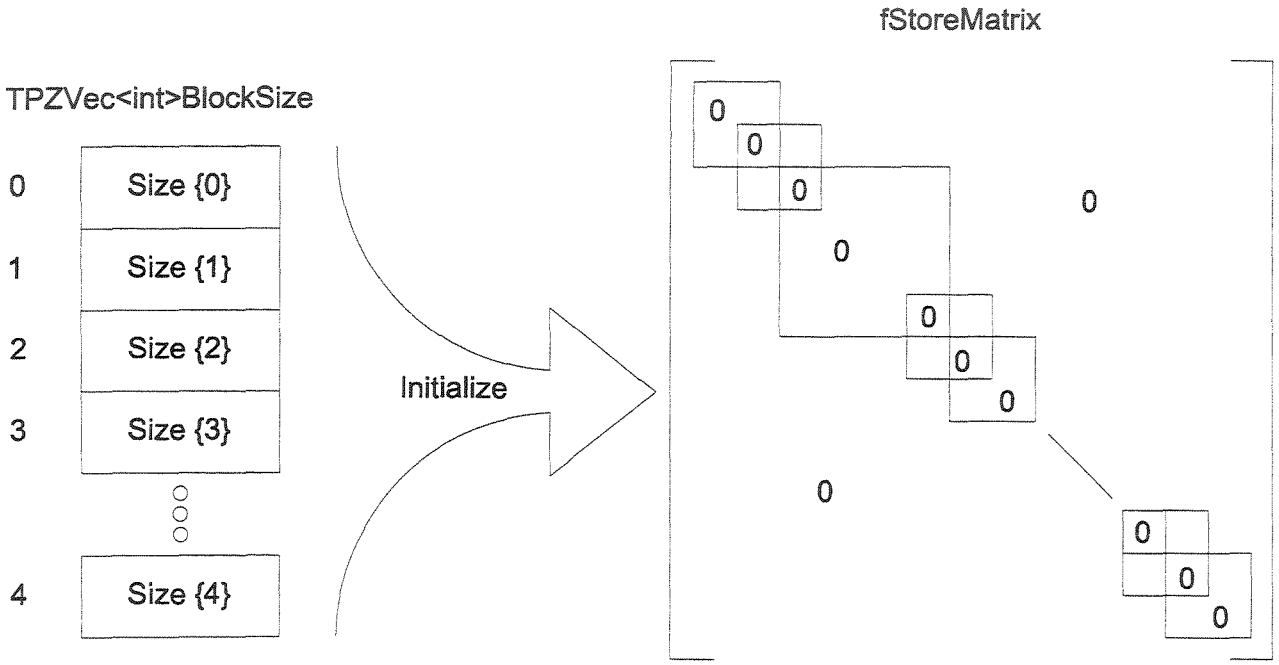


Figura B.3: Inicialização dos blocos

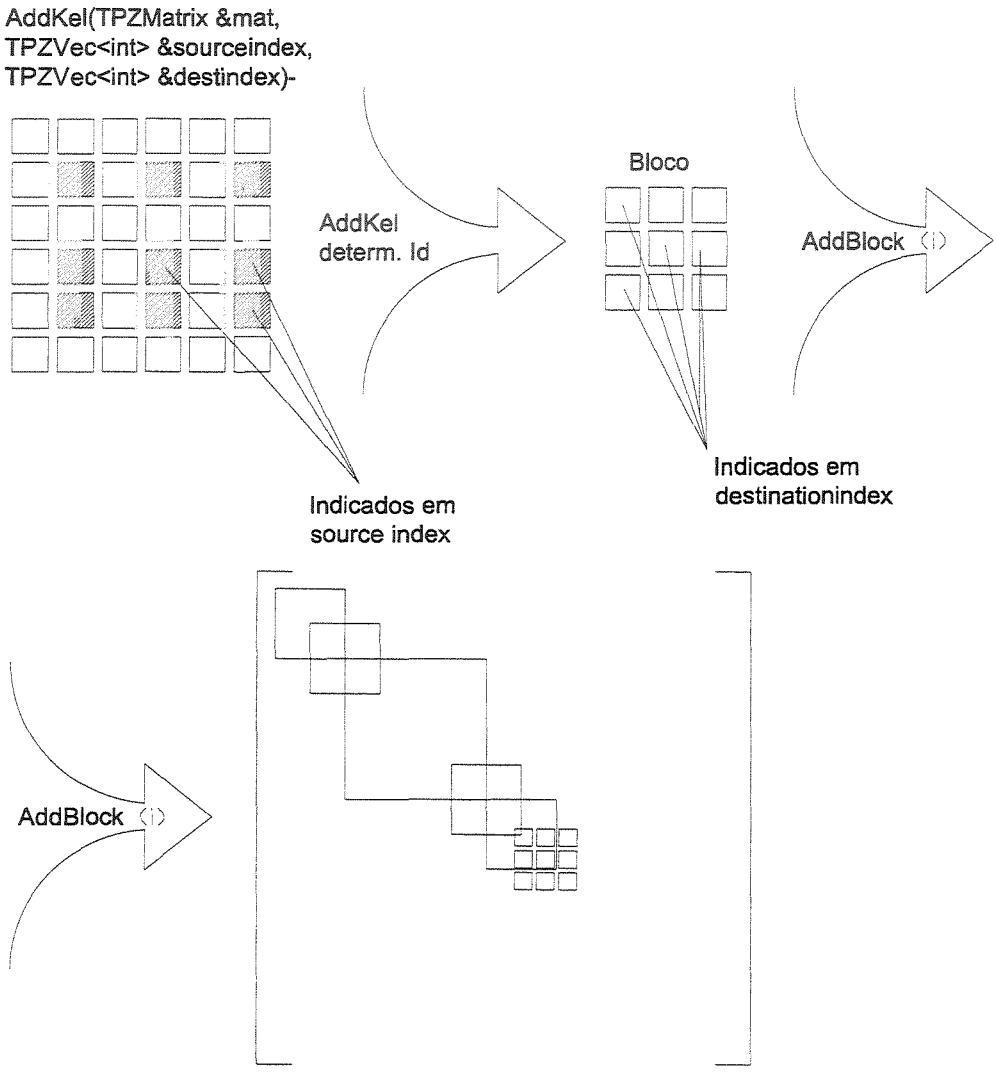
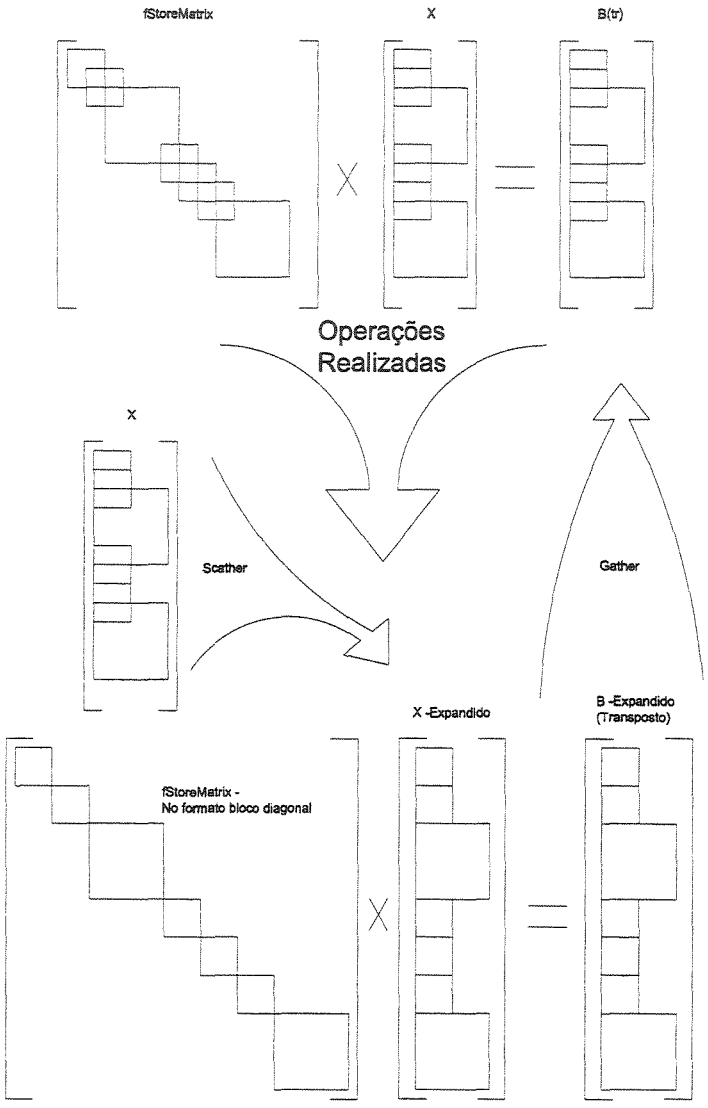


Figura B.4: Multiplicação



Cria uma cópia de uma matriz bloco sobreposto passada como parâmetro.

~TPZOverlapBlock()

Destrutor simples, não existem objetos alocados dinamicamente para apagar da memória, não sendo assim esta função utilizada.

Initialize(TPZVec<int> blocksize)

Através da dimensão do vetor identifica o número de blocos a ser criado, onde cada bloco terá a dimensão especificada pelos elementos de *blocksize*.

Métodos Virtuais

virtual REAL s(int row, int col)

Verifica se existe elemento da matriz em uma posição $(i, j) = (row, col)$

int Dim()

Retorna a dimensão da matriz (considera-se que a matriz é quadrada).

int Zero()

Define como zero todos os elementos da matriz.

void Transpose(TPZMatrix &T)

Retorna em T a matriz transposta do atual objeto.

int Decompose_LU()

Utiliza a fatoração LU para o cálculo da matriz triangular superior.

virtual int Substitution(TPZMatrix &B)

Realiza a retrosubstituição para a obtenção da solução do problema $A.x = b$, onde A já é uma matriz triangular superior.

void AddKel(TPZFMatrix &mat, TPZVec<int> &sourceindex, TPZVec<int> &destinationindex)

Realiza a inserção de uma matriz relativa a matriz de rigidez de um elemento, com restrições, em *mat*, cujos índices do elemento estão em *sourceindex* e os índices relativos à matriz global estão em *destinationindex*.

Com estes dados cria-se um bloco com os elementos referenciados, sendo este bloco então passado ao método *AddKel* descrito abaixo.

void AddKel(TPZFMatrix &mat, TPZVec<int> &destination)

Esse método calcula o índice do bloco e passa esse índice e os parâmetros de entrada para o método *AddBlock* realizar a inserção do bloco na matriz.

Métodos Específicos

void AddBlock(int id, TPZVec<int> &pos, TPZFMatrix &block)

Dado o índice identificador do bloco - *id*, o vetor com a posição de cada elemento do bloco - *pos*, em uma "matriz global" e o bloco *block* este é armazenado em *fStoreMatrix*.

void GetBlock(ind id, TPZFMatrix &block)

Retorna em block o bloco de índice *id*.

void Scatter(TPZFMatrix &uin, TPZFMatrix &uot, int stride)

Esse método cria um vetor de carga, que pode ter *"stride"* variáveis de estado, baseado em índices dos blocos através de um vetor com *" stride"* variáveis de estado baseado em índices

de conectividades.

void Gather(TPZFMatrix &fin, TPZFMatrix &fot, int stride)

Consiste na operação inversa de *Scatter*, onde dado um vetor baseado em índices de blocos, este é condensado em um vetor baseado em índices de conectividades, onde as contribuições de todos os blocos para uma conectividade são levados em conta.

void MultAdd(TPZFMatrix x, y, z, REAL alpha, beta, int opt, stride)

Realiza o seguinte cálculo:

$$z = alpha.x + beta.y.(opt.fStoreMatrix)$$

O procedimento para realização do cálculo é o representado no algoritmo descrito acima, onde é feita a operação *Scatter* em x, possibilitando a multiplicação bloco a bloco e posterior operação *Gather* para condensação do resultado.

Apêndice C

Reimplementação de Mudança de Coordenadas no PZ

O sistema previamente implementado no PZ estava capacitado a realizar transformação de coordenadas entre sistemas cilíndricos e cartesianos com mesma origem e eixos z coincidentes.

Aqui propôs-se a tornar essas alterações genéricas, aumentando a gama de possibilidades de mudança de coordenadas dentro do ambiente PZ, possibilitando a implementação de sistemas de coordenadas cilíndricos, com origem e eixos não coincidentes com os eixos de referência do sistema de coordenadas global e também implementação do sistema de coordenadas esféricas.

C.1 Desenvolvimento Teórico

A generalização dos métodos existentes implica na abstração seguinte: considerando que a estrutura existente está vinculada à um sistema de coordenadas com origem {x=0,y=0,z=0}, podemos considerar que tal origem refere-se a um sistema de coordenadas local, que por sua vez será o resultado de um conjunto de rotação e translação do sistema de referência global.

Assim, a partir do sistema denominado local, pode ser realizado um mapeamento através de um tensor composto por um tensor Translação [T] e um tensor rotação [R].

O tensor rotação, dadas as rotações em torno de cada eixo, pode ser escrito da seguinte forma:

$R = B.C.D$, onde:

$$B = \begin{bmatrix} cos\psi & sen\psi & 0 \\ -sen\psi & cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}, C = \begin{bmatrix} cos\theta & 0 & sen\theta \\ 0 & 1 & 0 \\ -sen\theta & 0 & cos\theta \end{bmatrix}, D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\phi & sen\phi \\ 0 & -sen\phi & cos\phi \end{bmatrix}$$

onde:

ψ: rotação em torno do eixo z;

θ: rotação em torno do eixo y;

ϕ: rotação em torno do eixo x;

O determinante do Jacobiano será nesse caso sempre igual a 1, pois essa transformação não implica em deformação do elemento.

Feita a translação e a rotação de um sistema cartesiano, conforme mostrado, a transformação para coordenadas cilíndricas ou esféricas consiste simplesmente na aplicação do Jacobiano da transformação sobre o sistema anterior.

Assim, para o caso de coordenadas cilíndricas teríamos:

$x = r * cos(\theta)$;

$y = r * sen(\theta)$;

$z = z$;

$$J = \begin{bmatrix} cos\theta & -r * sen\theta & 0 \\ sen\theta & r * cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ e } det(J) = r$$

Assim, podemos realizar a transformação de coordenadas:

$dx.dy.dz = det(J).dr.d\theta.dz$

Procedimento análogo pode ser feito para o sistema esférico ou qualquer outro sistema.

C.2 Implementação no PZ

No ambiente atual, todas as transformações são feitas na classe responsável pelo mapeamento geométrico do elemento. No decorrer do trabalho verificou-se que tal tarefa deveria ser realizada na Classe *TPZCosys*.

Assim, são desenvolvidas as seguintes implementações na classe *TPZCosys*:

- implementação de método para o cálculo do jacobiano de cada transformação;
- implementação de métodos para mapeamento translação;
- implementação de métodos para mapeamento rotação;
- implementação de método para transformação de coordenadas do sistema atual em cartesianas e vice-versa;

C.2.1 Mapeamento Translação

O Mapeamento translação consiste na mudança da origem do sistema de coordenadas para um outro ponto dado. Os eixos do sistema local obtido serão paralelos aos eixos do sistema global.

Em termos numéricos, a integral de uma função no novo sistema de coordenadas será dada pela integral da função em relação às coordenadas locais, acrescida da parcela relativa à região entre o sistema local e o sistema global.

C.2.2 Mapeamento Rotação

O mapeamento rotação consiste na mudança de coordenadas a partir de rotações nos eixos do sistema de coordenadas de referência. Essa rotação, nesse trabalho será representada através dos ângulos de Euler (ψ, θ, ϕ), representando o giro dado nos eixos x, y e z respectivamente.

Dados os ângulos de Euler, através do tensor mostrado em C.1, temos o mapeamento do sistema de coordenadas global para o sistema de coordenadas local. A obtenção da transformação inversa, para esse caso, é simples, pois o tensor rotação tem a propriedade de que seu inverso é igual ao seu transposto.

Com os dados do tensor também é possível o cálculo do Jacobiano da transformação e de seu determinante cujo valor é sempre igual a 1.

C.3 Implementação de Transformação de Coordenadas Cartesianas para Cilíndricas e Vice-Versa

Tais métodos encontram-se implementados na classe *TPZCylin*, tendo sido implementado um método para o cálculo do jacobiano da transformação e o cálculo de seu determinante.

Também foi implementado um ponteiro para o sistema de coordenadas de referência, para essa transformação, de modo a possibilitar a mudança de coordenadas inversa.

C.3.1 Documentação do Código Gerado

A documentação do código gerado pode ser encontrada no endereço `http://labmec.fec.unicamp.br/~pz/doxygen/class_TPZCosys.html`.

Toda a documentação gerada nesse trabalho foi gerada através do programa *Doxygen*, que identifica determinados caracteres no código e os interpreta como notas de documentação, tornando possível a geração da documentação em diversos formatos tais como: *latex*, *rich text format*, *html* e *man*.

Acoplado ao *Doxygen* está sendo utilizado o módulo *DOT*, da *AT&T*, que cria diagramas de interconexão entre classes, mostrando os parâmetros passados entre as classes conectadas.

Apêndice D

Subestruturação

Esse trabalho insere-se no desenvolvimento do ambiente de programação orientada a objetos PZ [16], para a resolução de problemas de valor de contorno através do método dos elementos finitos.

A implementação da subestruturação no PZ tem os seguintes objetivos principais:

1. Possibilitar a resolução de submalhas através da utilização de processamento paralelo;

2. Aumentar a flexibilidade para a análise.

A utilização de subestruturação para a resolução de problemas de elementos finitos é intuitiva e de implementação relativamente simples, dado que o procedimento consiste no envio de uma subestrutura para cada processador, sendo ao final da resolução de cada subestrutura, enviada a próxima subestrutura da fila, até que todas tenham sido resolvidas, sendo então necessário apenas a remontagem do sistema global.

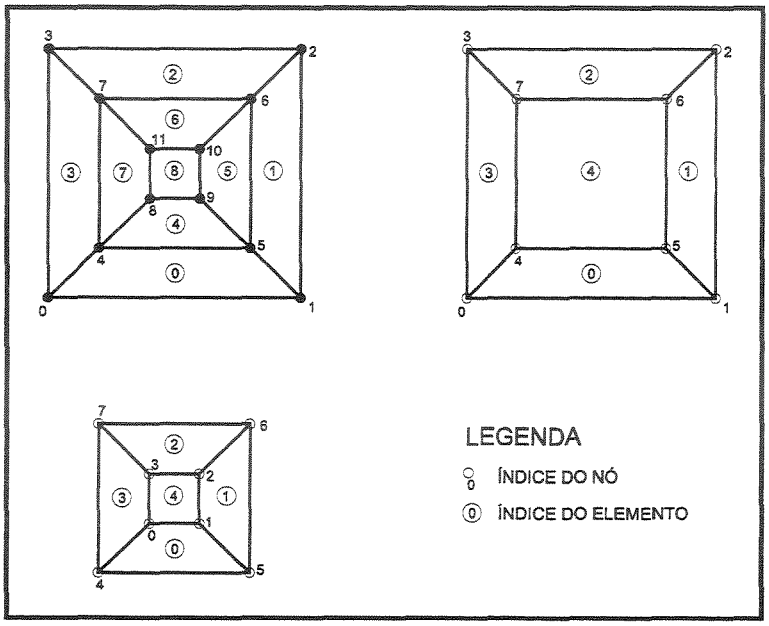
No caso de utilização de processamento paralelo, a solução de problemas com grande número de equações é feita em menos tempo, viabilizando assim o uso do método.

Dado o custo computacional para o gerenciamento das submalhas e do processamento, a adoção dessa abordagem em problemas de menor porte não se mostrará interessante.

Desse modo, a implementação de procedimentos de subestruturação confiáveis, representam um grande salto na busca de um ambiente computacional de elementos finitos com uso prático, o qual pode apresentar resultados rápidos e confiáveis para problemas reais, de alta complexidade.

Outro aspecto interessante da subestruturação é a possibilidade da utilização de diferentes métodos de análise para cada sub-malha, podendo em um determinado trecho ser utilizada a resolução do sistema através de métodos com eficiência comprovada para aquele tipo específico de matriz, utilizando os métodos convencionais, tais como LU, apenas nas regiões do domínio onde for necessário o uso de métodos genéricos.

Figura D.1: Exemplo de Subestruturação



D.1 Visão de álgebra linear

Para a utilização da subestruturação há a necessidade de se encontrar um sistema linear de equações, equivalente ao sistema original, no qual o número de equações é menor estando a contribuição dos elementos internos do subdomínio expresso através dos nós de interface entre o subdomínio e a malha inicial.

Para o melhor entendimento, consideremos a primeira malha representada na Figura (D.1, pág. 134), com 9 elementos e 12 nós.

Considerando, por simplicidade, que cada nó tem apenas um grau de liberdade, a matriz de rigidez e o vetor de carga dessa malha podem ser representados pelo sistema da Equação (D.1) (pág.135).

$$\begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} & k_{0,4} & k_{0,5} & k_{0,6} & k_{0,7} & k_{0,8} & k_{0,9} & k_{0,10} & k_{0,11} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} & k_{1,4} & k_{1,5} & k_{1,6} & k_{1,7} & k_{1,8} & k_{1,9} & k_{1,10} & k_{1,11} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} & k_{2,4} & k_{2,5} & k_{2,6} & k_{2,7} & k_{2,8} & k_{2,9} & k_{2,10} & k_{2,11} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} & k_{3,4} & k_{3,5} & k_{3,6} & k_{3,7} & k_{3,8} & k_{3,9} & k_{3,10} & k_{3,11} \\ k_{4,0} & k_{4,1} & k_{4,2} & k_{4,3} & k_{4,4} & k_{4,5} & k_{4,6} & k_{4,7} & k_{4,8} & k_{4,9} & k_{4,10} & k_{4,11} \\ k_{5,0} & k_{5,1} & k_{5,2} & k_{5,3} & k_{5,4} & k_{5,5} & k_{5,6} & k_{5,7} & k_{5,8} & k_{5,9} & k_{5,10} & k_{5,11} \\ k_{6,0} & k_{6,1} & k_{6,2} & k_{6,3} & k_{6,4} & k_{6,5} & k_{6,6} & k_{6,7} & k_{6,8} & k_{6,9} & k_{6,10} & k_{6,11} \\ k_{7,0} & k_{7,1} & k_{7,2} & k_{7,3} & k_{7,4} & k_{7,5} & k_{7,6} & k_{7,7} & k_{7,8} & k_{7,9} & k_{7,10} & k_{7,11} \\ k_{8,0} & k_{8,1} & k_{8,2} & k_{8,3} & k_{8,4} & k_{8,5} & k_{8,6} & k_{8,7} & k_{8,8} & k_{8,9} & k_{8,10} & k_{8,11} \\ k_{9,0} & k_{9,1} & k_{9,2} & k_{9,3} & k_{9,4} & k_{9,5} & k_{9,6} & k_{9,7} & k_{9,8} & k_{9,9} & k_{9,10} & k_{9,11} \\ k_{10,0} & k_{10,1} & k_{10,2} & k_{10,3} & k_{10,4} & k_{10,5} & k_{10,6} & k_{10,7} & k_{10,8} & k_{10,9} & k_{10,10} & k_{10,11} \\ k_{11,0} & k_{11,1} & k_{11,2} & k_{11,3} & k_{11,4} & k_{11,5} & k_{11,6} & k_{11,7} & k_{11,8} & k_{11,9} & k_{11,10} & k_{11,11} \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \\ a_8 \\ a_9 \\ a_{10} \\ a_{11} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \end{bmatrix} \tag{D.1}$$

Considerando que os elementos de 4 a 8 formam agora uma sub-malha, temos que seu sistema de equações é aquele mostrado em D.2.

$$\begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} & k_{0,\overline{4}} & k_{0,\overline{5}} & k_{0,\overline{6}} & k_{0,\overline{7}} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} & k_{1,\overline{4}} & k_{1,\overline{5}} & k_{1,\overline{6}} & k_{1,\overline{7}} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} & k_{2,\overline{4}} & k_{2,\overline{5}} & k_{2,\overline{6}} & k_{2,\overline{7}} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} & k_{3,\overline{4}} & k_{3,\overline{5}} & k_{3,\overline{6}} & k_{3,\overline{7}} \\ k_{\overline{4},0} & k_{\overline{4},1} & k_{\overline{4},2} & k_{\overline{4},3} & k_{\overline{4},\overline{4}} & k_{\overline{4},\overline{5}} & k_{\overline{4},\overline{6}} & k_{\overline{4},\overline{7}} \\ k_{\overline{5},0} & k_{\overline{5},1} & k_{\overline{5},2} & k_{\overline{5},3} & k_{\overline{5},\overline{4}} & k_{\overline{5},\overline{5}} & k_{\overline{5},\overline{6}} & k_{\overline{5},\overline{7}} \\ k_{\overline{6},0} & k_{\overline{6},1} & k_{\overline{6},2} & k_{\overline{6},3} & k_{\overline{6},\overline{4}} & k_{\overline{6},\overline{5}} & k_{\overline{6},\overline{6}} & k_{\overline{6},\overline{7}} \\ k_{\overline{7},0} & k_{\overline{7},1} & k_{\overline{7},2} & k_{\overline{7},3} & k_{\overline{7},\overline{4}} & k_{\overline{7},\overline{5}} & k_{\overline{7},\overline{6}} & k_{\overline{7},\overline{7}} \end{bmatrix} * \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_{\overline{4}} \\ a_{\overline{5}} \\ a_{\overline{6}} \\ a_{\overline{7}} \end{bmatrix} = \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_{\overline{4}} \\ u_{\overline{5}} \\ u_{\overline{6}} \\ u_{\overline{7}} \end{bmatrix} \tag{D.2}$$

O traço sobrescrito ao índice do elemento indica que este sofrerá contribuição da sub-malha.

Já a sub-malha terá o sistema mostrado abaixo.

$$\begin{bmatrix} k_{\underline{4},\underline{4}} & k_{\underline{4},\underline{5}} & k_{\underline{4},\underline{6}} & k_{\underline{4},\underline{7}} & k_{\underline{4},8} & k_{\underline{4},9} & k_{\underline{4},10} & k_{\underline{4},11} \\ k_{\underline{5},\underline{4}} & k_{\underline{5},\underline{5}} & k_{\underline{5},\underline{6}} & k_{\underline{5},\underline{7}} & k_{\underline{5},8} & k_{\underline{5},9} & k_{\underline{5},10} & k_{\underline{5},11} \\ k_{\underline{6},\underline{4}} & k_{\underline{6},\underline{5}} & k_{\underline{6},\underline{6}} & k_{\underline{6},\underline{7}} & k_{\underline{6},8} & k_{\underline{6},9} & k_{\underline{6},10} & k_{\underline{6},11} \\ k_{\underline{7},\underline{4}} & k_{\underline{7},\underline{5}} & k_{\underline{7},\underline{6}} & k_{\underline{7},\underline{7}} & k_{\underline{7},8} & k_{\underline{7},9} & k_{\underline{7},10} & k_{\underline{7},11} \\ k_{8,\underline{4}} & k_{8,\underline{5}} & k_{8,\underline{6}} & k_{8,\underline{7}} & k_{8,8} & k_{8,9} & k_{8,10} & k_{8,11} \\ k_{9,\underline{4}} & k_{9,\underline{5}} & k_{9,\underline{6}} & k_{9,\underline{7}} & k_{9,8} & k_{9,9} & k_{9,10} & k_{9,11} \\ k_{10,\underline{4}} & k_{10,\underline{5}} & k_{10,\underline{6}} & k_{10,\underline{7}} & k_{10,8} & k_{10,9} & k_{10,10} & k_{10,11} \\ k_{11,\underline{4}} & k_{11,\underline{5}} & k_{11,\underline{6}} & k_{11,\underline{7}} & k_{11,8} & k_{11,9} & k_{11,10} & k_{11,11} \end{bmatrix} * \begin{bmatrix} a_{\underline{4}} \\ a_{\underline{5}} \\ a_{\underline{6}} \\ a_{\underline{7}} \\ a_8 \\ a_9 \\ a_{10} \\ a_{11} \end{bmatrix} = \begin{bmatrix} u_{\underline{4}} \\ u_{\underline{5}} \\ u_{\underline{6}} \\ u_{\underline{7}} \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \end{bmatrix} \tag{D.3}$$

De modo similar aquilo que ocorre na malha pai, a sub-malha tem elementos que contribuem também na malha pai. Esses elementos foram caracterizados com um traço subscrito no índice do nó de interface.

Para que o método mostre-se utilizável, devemos demonstrar que:

1. Há uma forma de expressar a contribuição dos nós internos da sub-malha nos seus nós externos;

2. Os resultados para os dois sistemas são iguais;

3. Que dados os resultados dos nós de interface é possível obter-se os resultados nos nós internos da sub-malha.

Todos esses requisitos são preenchidos através do conceito de redução estática.

O sistema de equações utilizado na redução estática é mostrado no algoritmo (3) (pág. 136), onde as conectividades internas são representadas pelo índice "i" e as conectividades onde os graus de liberdade interno serão condensados através do índice "c" .

Algorithm 3 Matrizes Reduzidas

$$\begin{bmatrix} K_{ii} & Kic \\ K_{ci} & K_{cc} \end{bmatrix} * \begin{bmatrix} U_i \\ U_c \end{bmatrix} = \begin{bmatrix} F_i \\ F_c \end{bmatrix}$$

A partir do sistema inicial, a condensação dos graus de liberdade $[U_i]$ sobre os graus de liberdade $[U_c]$ é definida conforme o algoritmo (4) (pág. 136).

Algorithm 4 Processo de Condensação Estática

$$[K_{ii}]_{red} = [K_{cc}] - [K_{ci}] \cdot [K_{ii}]^{-1} \cdot [K_{ic}]$$
$$[f_c]_{red} = [f_c] - [K_{ci}] \cdot [K_{ii}]^{-1} \cdot [f_i]$$

Assim, o sistema reduzido torna-se aquele apresentado na equação (5) (pág. 136).

Algorithm 5 Sistema condensado

$$[K_{cc}]_{red} \cdot [U_c] = [f_c]_{red}$$

D.2 Visão de elementos finitos

Quando coloca-se o termo sub-malha, deve ficar claro que o domínio em estudo está sendo dividido e, de modo a compatibilizar as equações do subdomínio às equações do domínio global há a necessidade de se inserir o conceito de "nó interno".

Em uma sub-malha podem ser encontrados nós, ou conectividades, com as seguintes características:

- nó externo: um nó é considerado externo quando suas conectividades apresentarem elementos de objetos malha, ou sub-malha, distintos. Dessa forma, esse nó terá, no mínimo, um índice para a sub-malha analisada e um índice para a malha pai;

- nó dependente: um nó dependente tem como característica que suas equações estão vinculadas ao sistema de equações da malha pai. Assim, não é necessário que o elemento esteja locado na interface da sub-malha para ele comportar-se como nó externo. De fato, a menos que seja feita chamada específica para tornar os nós internos, todos as conectividades de uma sub-malha continuarão externos após a sua criação, pois eles guardarão as referências para as equações da malha pai de onde esta sub-malha foi criada;

- nó interno: pode se dizer que um determinado nó é interno, quando todas as suas conectividades pertencem ao mesmo objeto malha e quando suas equações não estão referenciadas pelo sistema de equações da malha pai.

De modo a facilitar o entendimento desses conceitos, seguindo a seqüência da Figura (D.2, pág 138), temos:

- No passo (0), inicia-se pela discretização do domínio, gerando a malha inicial (*root mesh*);

- Em (1), é criada uma sub-malha. Quando da criação da sub-malha, são indicados quais os elementos da malha pai que constituem a sub-malha, sendo esses transferidos para o novo sub-domínio. Entretanto, nesse ponto, as conectividades da sub-malha ainda tem referências para o sistema de equações da malha pai e, dessa forma, ainda são nós dependentes ou externos;

- Seguindo para (2), é executado a chamada para tornar todos os nós sem dependências externas ou nós limítrofes em nós internos. Tal procedimento será detalhado posteriormente. Ao final da execução do procedimento, os nós que só possuem dependências de elementos da malha são transformados em nós internos;

- Feito isso, conforme pode ser visto no passo (3), a malha pai passa a ser constituída por um grande elemento, em substituição aos antigos elementos, que passaram a fazer parte da sub-malha vista em (4).

Com essa abordagem torna-se possível que a malha pai veja a sub-malha como um de seus elementos enquanto a sub-malha apresenta internamente características de malha.

Com isso, todas as operações que envolvam a manipulação das equações dos sistemas de malhas e submalhas devem ser adaptadas, de modo que a malha pai receba através dos nós externos toda a contribuição dos nós internos da sub-malha.

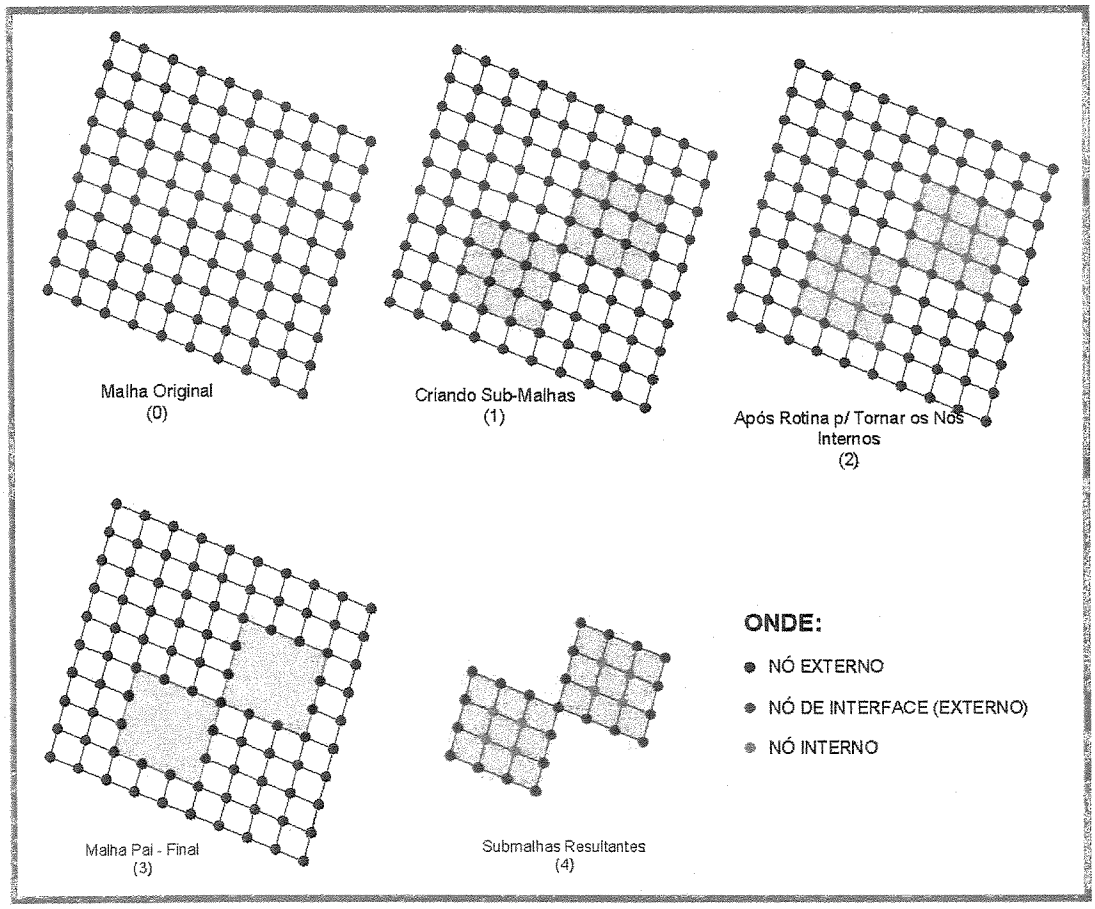


Figura D.2: Conceito de nó Interno

Para realizar tal procedimento é feito a condensação estática sobre os nós externos, de tal forma que a contribuição dos nós internos seja informada à malha pai através desses nós.

Por outro lado, todas as informações que são passadas da malha pai para a sub-malha precisam ser redistribuídas para os nós internos da sub-malha.

D.3 Descrição orientada para objetos dos conceitos de subestruturação

D.3.1 Redução estática de equações

A implementação da redução estática dos nós internos sobre os externos é feito no PZ através da Classe *TPZMatRed*.

A implementação realizada por Devloo e Mandujano [25], para sistemas de equações quadrados simétricos ou não, consiste na divisão das equações em dois conjuntos: um com as equações relativas às conectividades internas e outro relativo às externas. Também o vetor de carga é dividido em uma parte relativa 'as equações internas e outro relativa às equações das conexões externas.

No início da operação, os objetos desta classe comportam-se como objetos do tipo TPZ-Matrix, passando, após a redução do conjunto de equações internas sobre as externas, para um comportamento de matriz cheia condensada.

Assim, objetos dessa classe tem como principais características a existência de quatro matrizes onde, com exceção da matriz $[K_{II}]$, que é um ponteiro para um objeto do tipo TPZMatrix, as outras matrizes são ponteiros para objetos do tipo TPZFMatrix (matriz cheia).

Utilizando a metodologia descrita em 4, chegamos ao final do processo com as matrizes $[K_{ii}]_{red}$ e $[f_i]_{red}$ armazenadas nas próprias matrizes $[K_{ii}]$ e $[f_i]$, iniciais, sendo também armazenados os valores de $[K_{ii}]^{-1} \cdot [f_i]$ em $[f_i]$ e, em $[K_{ic}]$, os valores de $[K_{ii}]^{-1} \cdot [K_{ic}]$ [24].

Assim, o sistema reduzido torna-se aquele apresentado na equação abaixo:

$$[K_{cc}]_{red} \cdot [U_c] = [f_c]_{red}$$

D.3.2 Conceito de sub-malhas

Sob o aspecto de comportamento interno, uma sub-malha deve ter as mesmas características da malha, ou seja elementos, nós, conectividades, materiais, etc., além de ter uma malha "pai" e métodos que possibilitem a inserção e remoção de elementos e conectividades, a identificação da sua hierarquia etc.

Pelo aspecto da análise da malha pai, uma sub-malha deve ter comportamento de um elemento, pois nesse caso seu comportamento será de um "super-elemento", com matriz de rigidez e conectividades com os elementos vizinhos.

O ambiente PZ já tem implementadas classes de malha computacional, elementos computacionais, conectividades etc., sendo a classe gerada compatível com os métodos já implementados.

Utilizando a filosofia de programação orientada a objetos, tal problema é um caso típico de derivação, pois já estão implementadas as classes que servirão de base para o desenvolvimento.

D.4 Implementação da subestruturação

D.4.1 Uma classe matricial para redução estática

Conforme já mencionado, os nós internos da sub-malha contribuem na matriz de rigidez global através dos seus nós externos.

Para tal, é necessário realizar a condensação estática dos nós internos nos externos.

Como já demonstrado, tal processo é feito no PZ através dos objetos da classe *TPZMatRed*, sendo esta derivada das classes matriciais do PZ, *TPZMatrix* e *TPZFMatrix*.

Classes matriciais

As classes matriciais no PZ são todas derivadas da Classe *TPZMatrix*, na qual estão definidas as características básicas de objetos matriciais, sendo de fato uma classe abstrata, pois não são implementados em seu escopo todas as funções, sendo algumas definidas exclusivamente nas suas classes derivadas.

Assim, a classe *TPZMatrix* define apenas o comportamento que terão os objetos matriciais do PZ.

Já a classe *TPZFMatrix*, derivada da classe *TPZMatrix*, implementa o comportamento de matrizes cheias, sendo um grande número de operações matriciais implementados nessa classe.

Interface da classe matricial TPZMatRed

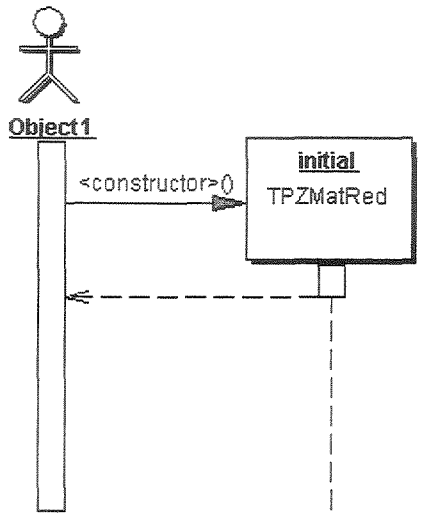
Para mostrar a forma como foi implementada essa classe serão utilizados os diagramas de seqüência que mostram o fluxo de operações realizadas em cada função, com descrição posterior dos diagramas.

Construtores

Existem dois construtores para essa classe, não sendo passado argumento para o construtor vazio. Assim é inicializado como um objeto vazio derivado de *TPZMatrix*, com suas matrizes e vetores também inicializados nulos (ver fig. D.3).

A outra forma de criar um objeto dessa classe é através da seguinte chamada:

Figura D.3: Construtor da Classe TPZMatRed



$$TPZMatRed(intdim, intdim00)$$

onde são passadas as dimensões do sistema inicial da sub-malha - dim - e a dimensão sob a qual este será reduzido estaticamente - dim00. Nessa inicialização, o objeto derivado de TPZMatrix é inicializado como uma matriz quadrada de dimensão "dim".

Destrutores

Todos os ponteiros para os vetores e matrizes da classes são removidos da memória.

O diagrama da Figura (D.4, pág. 142) mostra a seqüência de operações realizadas no destrutor da classe.

Definição dos Parâmetros

Como pode ser identificado, após criado o objeto *TPZMatRed*, esse só tem a definição de suas dimensões, sendo necessário definir os elementos componentes de vetores e matrizes. Os procedimentos para definição de dados são apresentados na seqüência.

O método *PutVal(int r, int c, REAL value)* indica o valor *value* que ocupará a posição coluna c e linha r.

O diagrama de seqüência desse método é apresentado na Figura (D.5, pág. 143).

O método *SetK00* define a matriz $[k_{00}]$, que deverá ser utilizada na condensação estática. A Figura (D.6, pág. 143) apresenta o diagrama de seqüência desse método.

Figura D.4: Destrutor da Classe TPZMatRed

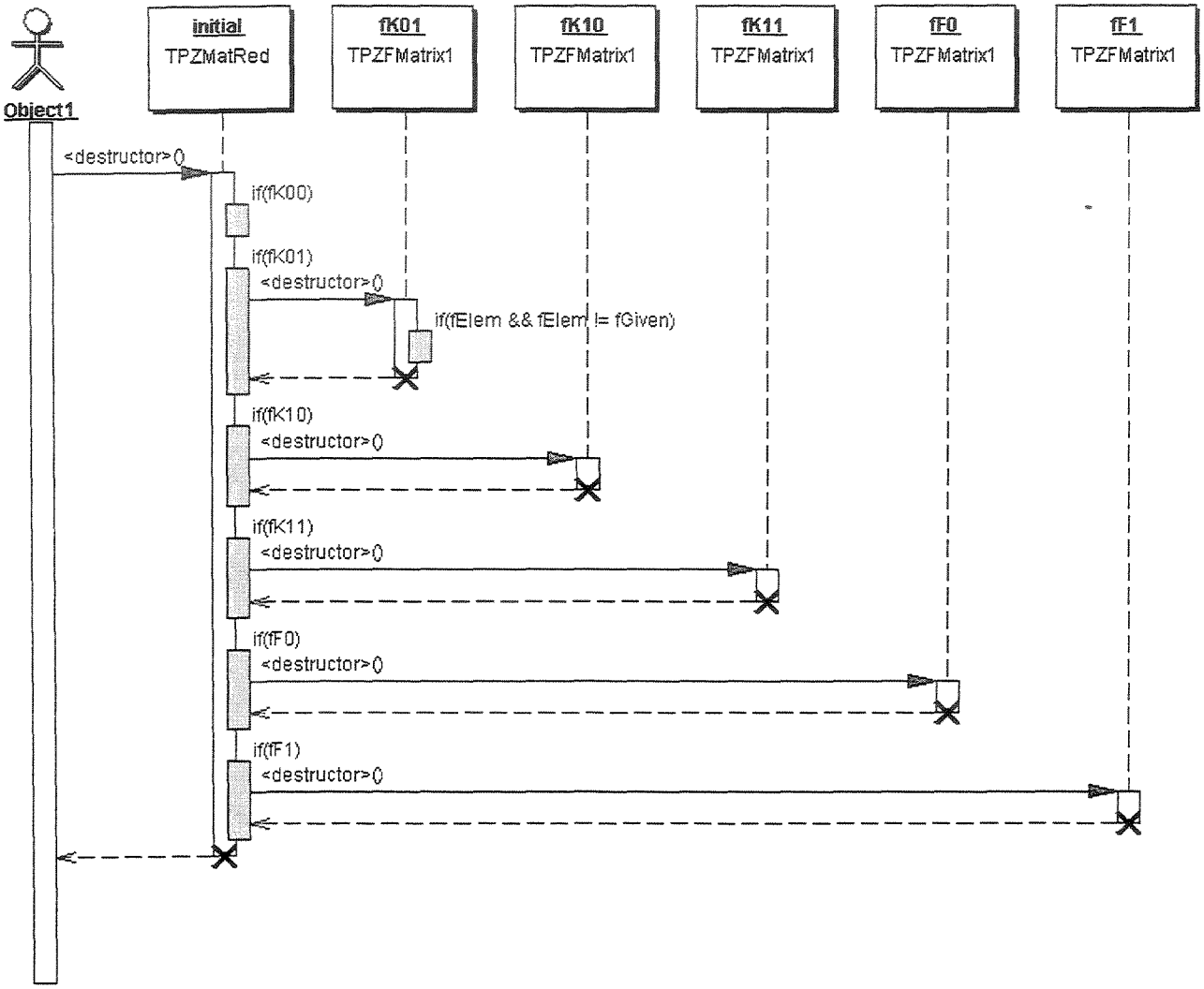


Figura D.5: Definição de Dados - PutVal

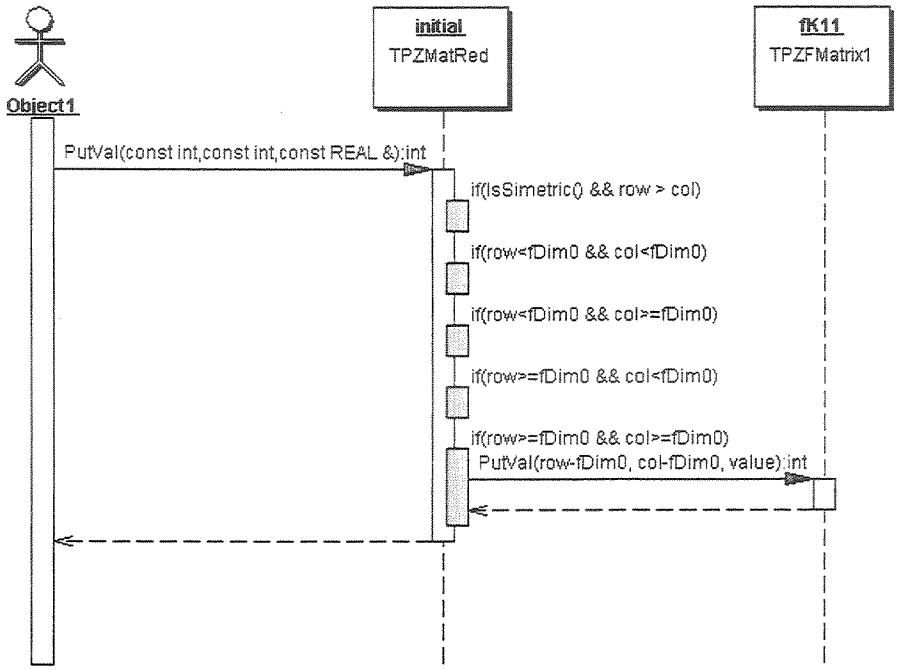


Figura D.6: Definição de Dados - SetK00

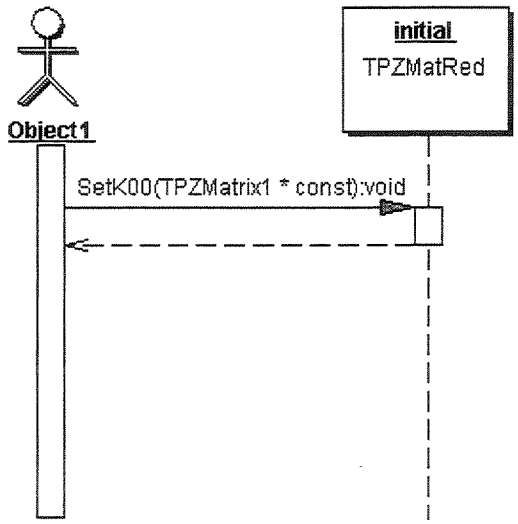
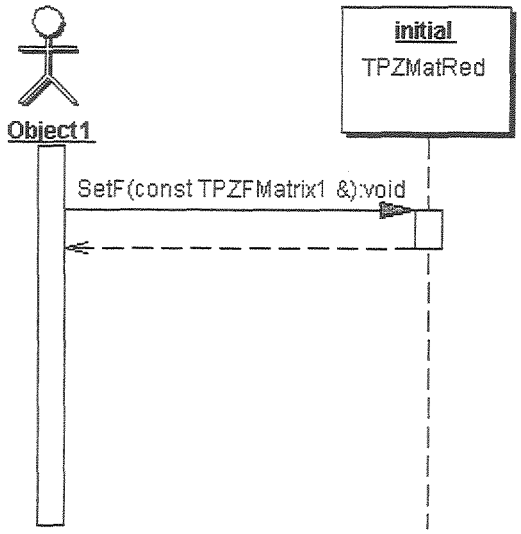


Figura D.7: Definição de Dados - SetF



O método *SetF* define o vetor de carga que será considerado na redução estática. O diagrama do método é mostrado na Figura (D.7, pág. 144).

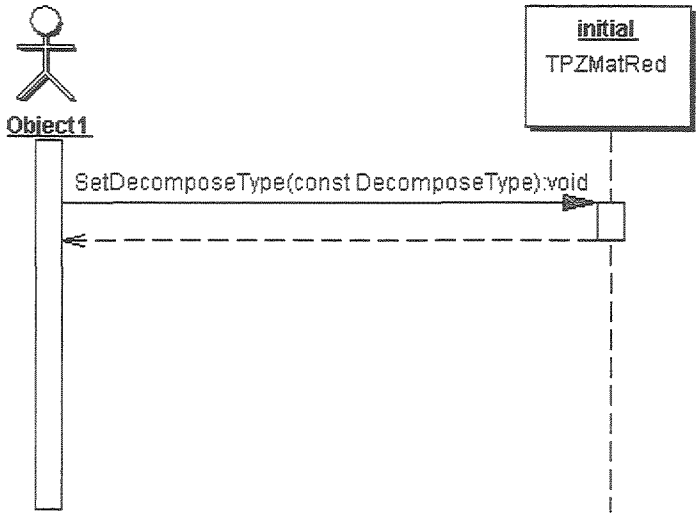
Realização dos Cálculos

Definidos as matrizes e vetores sob os quais serão realizadas as operações passa-se a efetivar a redução estática, para tal são definidas as seguintes funções.

Em primeiro lugar, pode-se definir o tipo de análise que será feita, podendo-se optar pelos seguintes tipos de análise:

- LU: decomposição padrão, a qual mostra-se uma forma robusta, pois pode ser utilizada para a resolução de todo o tipo de sistema linear;
- *Cholesky*: apresenta uma performance melhor que a LU, em termos de custo de processamento, entretanto só tem resultados garantidos caso seja aplicado a sistemas definidos positivos;
- *Frontal*: no caso de sistemas de grande porte apresenta uma performance melhor que os anteriores, pois reduz a utilização de memória, procurando-se armazenar os dados completamente no cache do processador e, conseqüentemente, reduzindo o tempo de processamento. Entretanto, para sistemas menores, essa abordagem não é satisfatória, pois o gerenciamento do processo para determinação da frente do sistema pode gerar um custo representativo.

Figura D.8: Cálculo - SetDecomposeType



- Métodos Paralelos: o PZ já tem implantados os métodos de análise descritos acima, para a utilização de processamento paralelo com memória compartilhada, entretanto, a restrição nesse caso é que se tenha um sistema multiprocessado.

A Figura (D.8, pág. 145) mostra o diagrama de seqüência da função de escolha do tipo de análise - *SetDecomposeType()*.

O método *F1Red()* calcula e devolve um ponteiro para o vetor de carga condensado.

A implementação básica do método consiste nos seguintes blocos:

- Verificação da consistência dos dados: verifica se existe a dimensão a que será reduzido estaticamente o sistema ou se o vetor de carga já foi reduzido. Caso qualquer um desses requisitos seja satisfeito, será apenas retornado um ponteiro para o vetor de carga reduzido existente;

- O bloco seguinte faz a verificação se a matriz de rigidez já foi reduzida, caso contrário o método de redução é ativado;

- Por fim são feitas as operações para a redução do vetor de carga.

A Figura (D.9, pág. 146) mostra o diagrama de seqüência da função *F11Red()*.

Da mesma forma que para o método *F11Red()*, o procedimento para o cálculo da matriz de rigidez condensada também passa pela verificação da consistência dos dados para posterior realização dos cálculos. O diagrama do método *K11Red()* é mostrado na Figura (147, pág. 147).

Figura D.9: Cálculo - F1Red

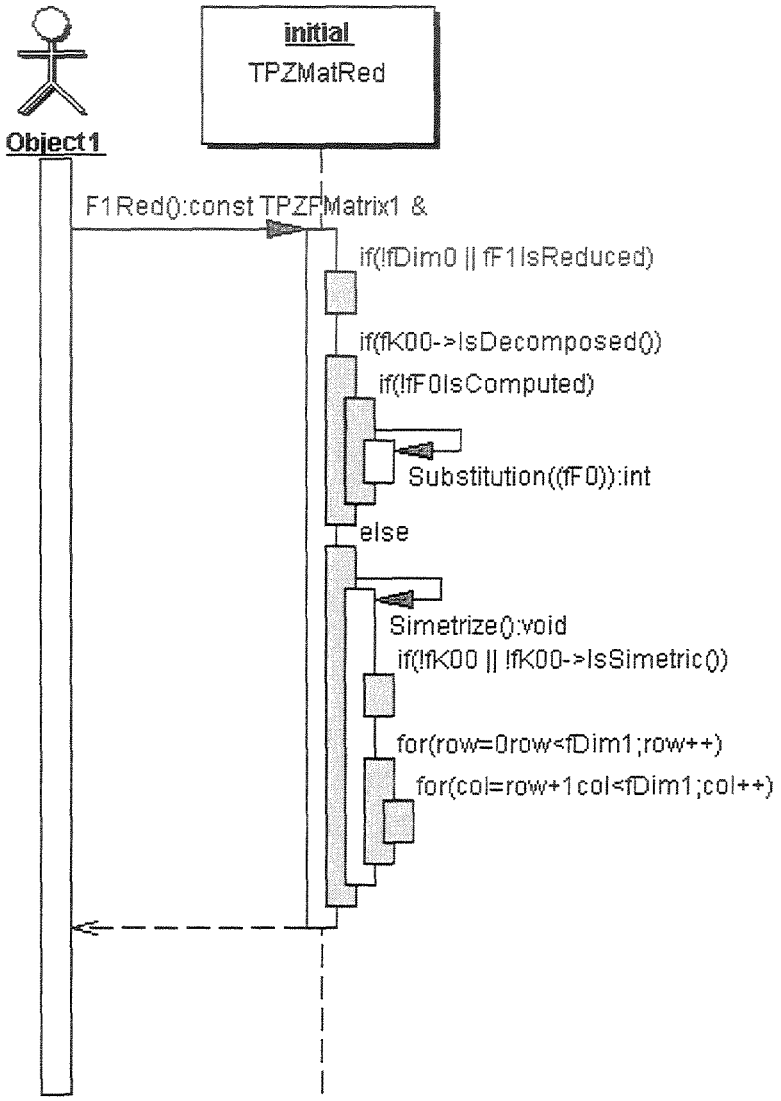


Figura D.10: Cálculo - K11Red

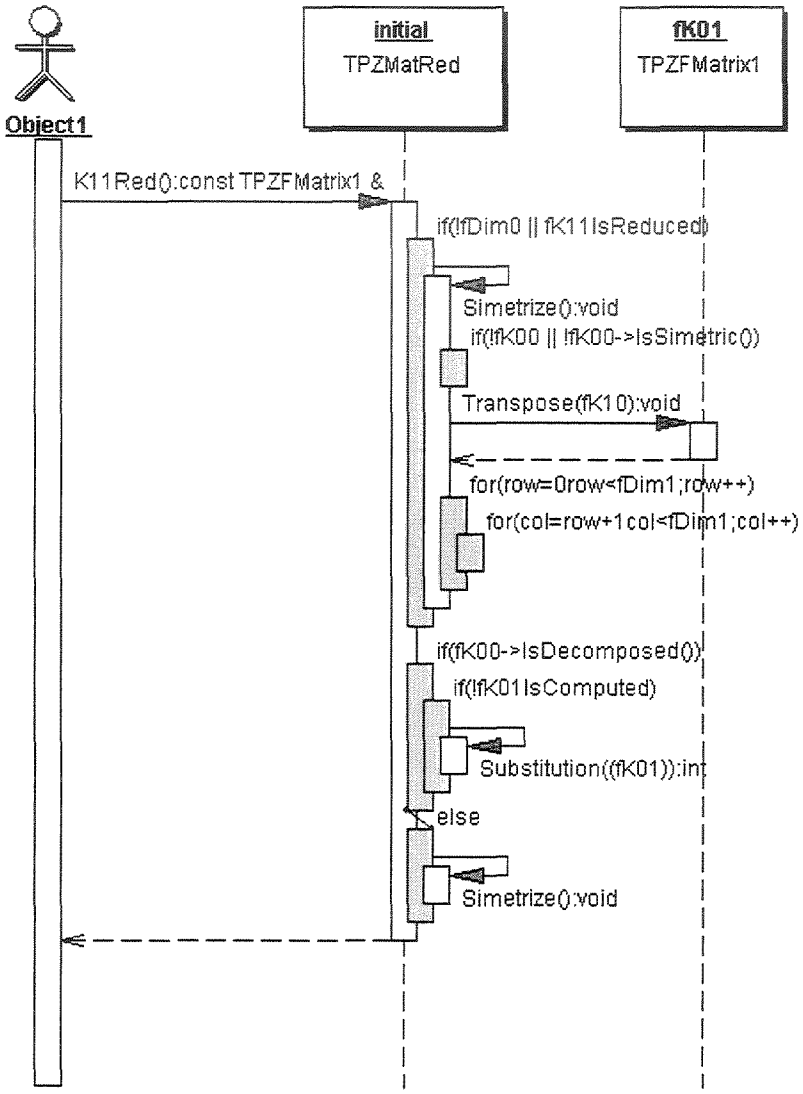
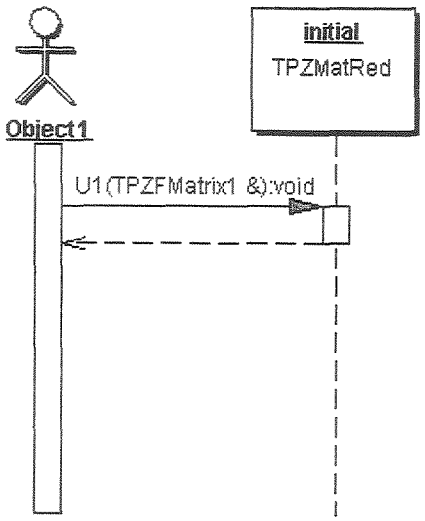


Figura D.11: Cálculo - U1()



O diagrama da Figura (D.11, pág. 148) mostra o cálculo de *U1*.

A seqüência de operações para o cálculo de *UGlobal* é apresentada na Figura (D.12, pág. 149).

Métodos Acessórios

Esses métodos são aqueles definidos como puramente virtuais na classe pai - *TPZMatrix*, além de métodos internos recorrentes.

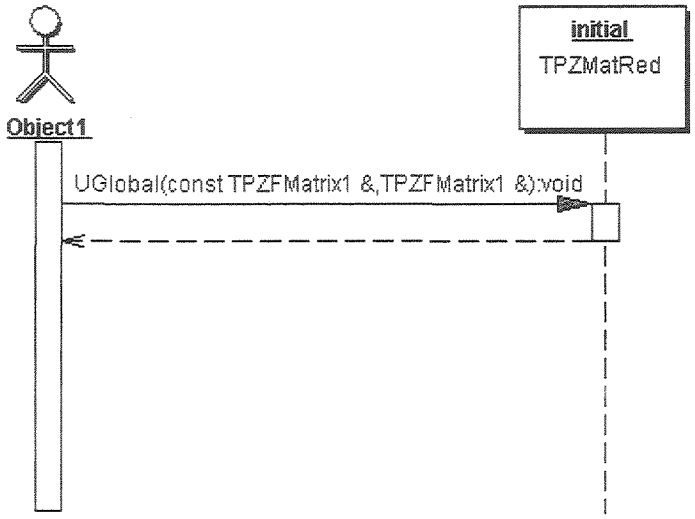
Dentre esses métodos destacam-se:

- *Zero()*: preenche todas as variáveis matriz e vetor do objeto com zero, além de redefinir todas as variáveis booleanas de modo a refletir que o objeto foi zerado;

- *Redim(int dim00, int dim)*: como a classe tem um construtor vazio, há a necessidade de que um método possibilite a definição posterior das dimensões das matrizes e vetores componentes do objeto. Esse procedimento é mostrado no diagrama de seqüência do método, mostrado na Figura (D.13, pág. 150);

- *Substitution (TPZMatrix *B)*: método utilizado durante a resolução do sistema. Sua implementação é mostrada na Figura (D.14, pág. 151).

Figura D.12: Cálculo - UGlobal



- *Print(char * name, ostream &out, MatrixOutPutFormat form)*: definições para impressão dos dados da classe, tais como nome do objeto - *name*, local de impressão - *out* e padrão de impressão - *form*;
- *Error (char *msg, char *msg2)*: gerenciador de erros.

D.4.2 Uma sub-malha como derivação dupla

Como já descrito anteriormente, a sub-malha precisa ter ora comportamento de malha e ora comportamento de elemento.

Dessa forma, seus métodos derivados precisam ser adaptados de modo a compatibilizar tal situação e poder ter o comportamento correto em cada momento que os objetos da classe são acessados.

Atribuições da classe TPZCompMesh

A classe *TPZCompMesh* contém informações sobre os elementos e nós computacionais da malha, bem como de suas condições de contorno . É neste objeto que ocorre o cálculo discreto dos elementos finitos.

As principais características dos objetos desta classe são:

- Acesso à malha discreta;
- Acesso aos dados do sistema de equações, tais como:

Figura D.13: Métodos Acessórios - Redim

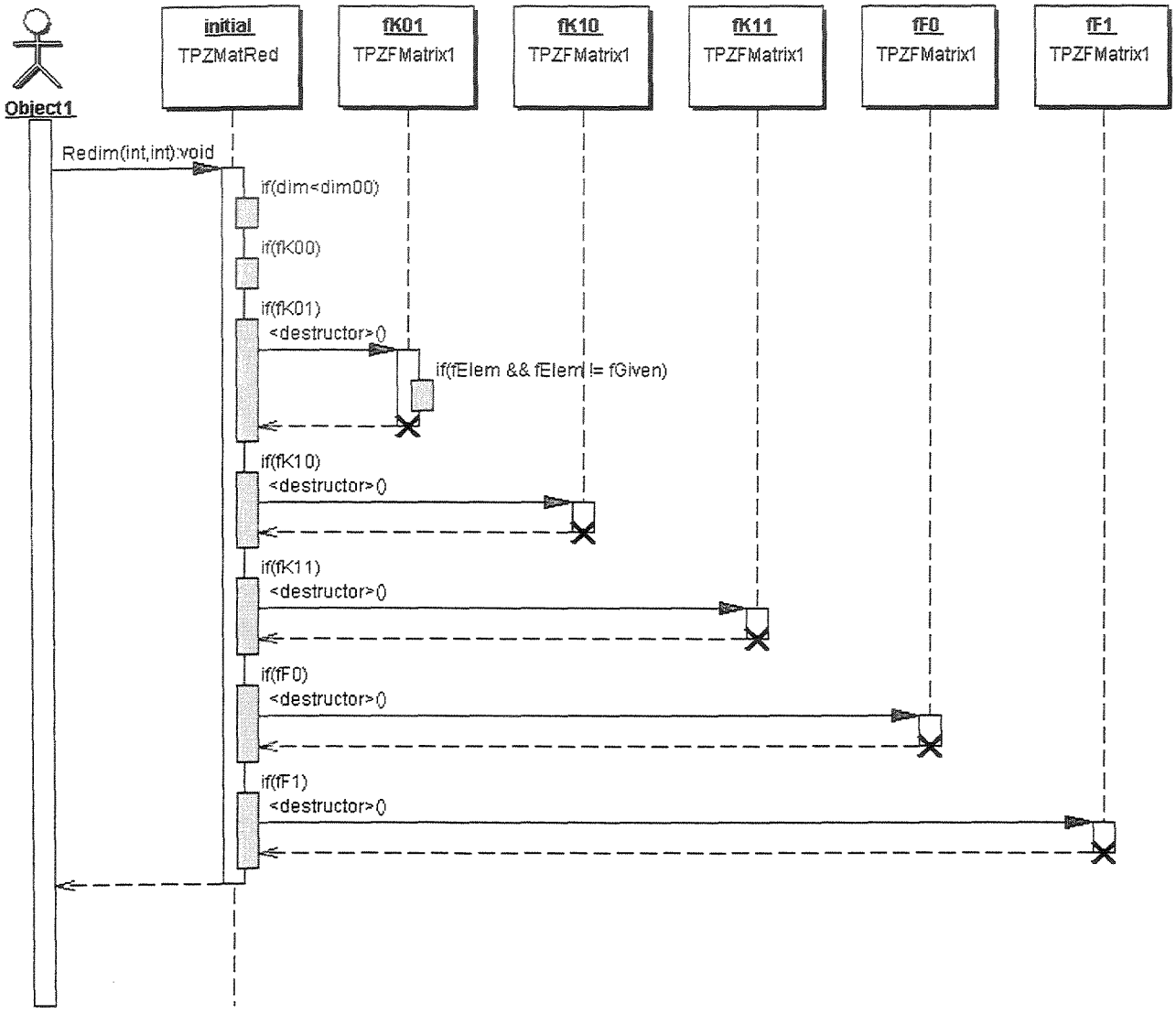
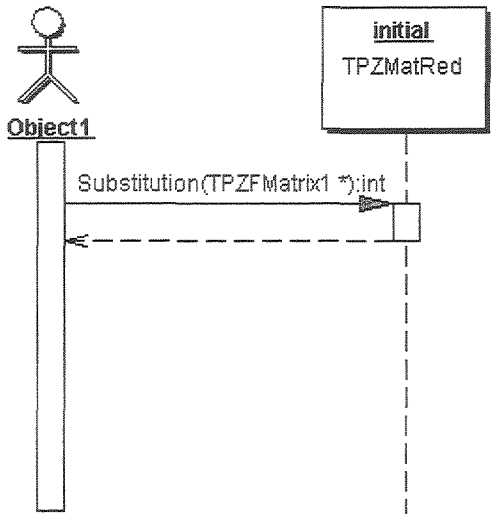


Figura D.14: Métodos Acessórios - Substitution



  - largura da banda,
  - número de equações;

- Renumeração dos nós, através dos algoritmos *Metis* ([17]) ou *Sloan*;
- Cálculo das dimensões dos blocos dos sistema de equações;
- Montagem do sistema de equações globais e do vetor de carga;
- Distribuição do vetor solução sobre os graus de liberdade (LoadSolution);

No ambiente PZ uma malha é implementada através de duas entidades: uma malha geométrica e uma malha computacional, estando a malha geométrica com as características topológicas e a malha computacional com as informações necessárias a solução de um problema.

Desta forma, a malha computacional pode ser composta por apenas um conjunto de elementos mestres, onde cada elemento mestre representa um determinado tipo, ou conjunto, de elementos, tais como triangulares, quadráticos, bi-quadráticos etc.

Assim, o cálculo da matriz de rigidez de uma malha reduz-se ao cálculo da matriz de rigidez de um conjunto de elementos mestres e o cálculo do jacobiano de mapeamento entre cada elemento geométrico e seu respectivo elemento mestre.

Tal forma de abordagem, além de ser interessante sob o aspecto de encapsulamento em programação orientada a objetos, facilita outras abordagens, tais como aquelas relativas à

mudança de sistemas de coordenadas, onde o mapeamento também é feito através de um Jacobiano.

Outro aspecto interessante é a possibilidade de uma mesma malha geométrica poder ser referenciada por mais de uma malha computacional, possibilitando que malhas, com diferentes graus de refinamento, possam ter a mesma malha geométrica de referência. Isso facilita a adaptatividade h.

Atribuições da classe TPZCompEl

Esta classe define as características básicas de um elemento computacional, sendo uma classe abstrata, ou seja, parte de seus métodos são implementados apenas nas suas classes derivadas.

As principais características dos objetos derivados desta classes são:

- Definição da ordem de interpolação e cálculo da função de forma do elemento e do jacobiano de transformação para o elemento mestre, possibilitando o cálculo do valor da função de forma em um dado ponto paramétrico;
- Definição da regra de integração, sendo o valor padrão adotado como sendo a regra suficiente para integração do quadrado da função de forma;
- Cálculo da matriz de rigidez do elemento;
- Aplicação de condições de contorno sobre a matriz de rigidez;
- Projeção do fluxo associado à lei de conservação sobre o espaço de interpolação;
- Cálculo do erro, através da implementação do estimador de Zienkiewicz e Zhu.

Comportamento da sub-malha como malha

Dadas as características requeridas pelos objetos sub-malha, aqui propostos, há a necessidade que esses tenham um comportamento dúbio. Quando analisados do ponto de vista da malha pai, eles devem comportar-se como um elemento computacional dessa malha.

Já quando iniciamos qualquer operação interna à submalha, esta deve apresentar um comportamento de malha.

Assim, as características de um objeto malha são implementados da seguinte forma:

- Acesso aos dados do sistema de equações:
  - largura da banda: é calculada a largura da banda da sub-malha, com o mesmo procedimento utilizado pela malha pai, sem necessidade de qualquer reimplementação,

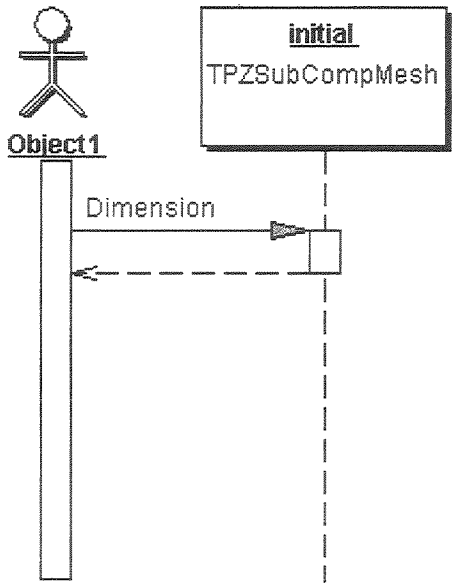


Figura D.15: Dimensão da sub-malha

  - número de equações: como o número de equações é obtido através das dimensões dos blocos dos elementos que constituem a malha, o procedimento para o cálculo do número de equações não é alterado em relação a definição virtual da classe pai;

- Renumeração dos nós: são utilizados os procedimentos virtuais da classe pai, os quais acessam os algoritmos Metis;

- Montagem do sistema de equações da sub-malha e do vetor de carga da sub-malha: também são utilizados os procedimentos já definidos na classe pai;

- Distribuição do vetor solução sobre os graus de liberdade (*LoadSolution*): como a sub-malha pode ter nós internos, cuja consideração no sistema global de equações é feito através da condensação estática dos nós internos sobre os nós externos da sub-malha, há a necessidade de quando do processo inverso, definir um procedimento para a distribuição do vetor de carga dos nós externos sobre os nós internos, tal procedimento é feito através do método *LoadSolution*.

Comportamento da sub-malha como elemento

Quando uma malha tem submalhas em sua estrutura, essas devem comportar-se como elementos, disponibilizando todas as informações inerentes a objetos desse tipo.

A principal informação que deve ser disponibilizada por um elemento é a sua matriz de rigidez, pois essa leva em consideração as características de materiais e disposição de nós

dentro da sub malha.

A Figura (D.16, pág. 155) apresenta o diagrama de seqüência da criação da matriz de rigidez da sub-malha.

A montagem da matriz de rigidez da sub-malha consiste em dois grupos de procedimentos

1. Criação da matriz de rigidez da sub-malha, considerando nós internos e externos;

2. Alocação da matriz de rigidez da sub-malha na matriz de rigidez da malha pai.

A criação da matriz de rigidez da sub-malha é feita da mesma forma que para qualquer malha, ou seja é feito de modo recursivo a montagem da matriz de rigidez de cada elemento e, na seqüência, é feita a contribuição da rigidez do elemento na matriz da sub-malha. Da mesma forma pode ser feito a montagem do vetor de carga da sub-malha.

Com a matriz da sub-malha montada é necessário fazer a contribuição desta na matriz da malha pai, sendo então necessário realizar o procedimento de redução estática dos nós internos sob os nós externos, seguindo a metodologia já descrita.

Dessa forma, realizando este procedimento de maneira recursiva, pode-se calcular a matriz de rigidez e o vetor de carga de uma malha com vários níveis de submalhas.

Transferência de um nó da malha pai para uma sub-malha

A transferência de nós da malha pai para a sub-malha envolve a utilização de algumas funções conforme descrito a seguir.

Função de Transferência de Nós da Malha Pai para a Sub-malha - GetFromSuperMesh(int superind, TPZCompMesh *supermesh)

Essa função realiza a transferência de um nó da malha pai para a sub-malha, realizando as seguintes tarefas:

- identificação da malha ancestral comum: toda a sub-malha tem uma malha pai e, extrapolando o conceito, a malha inicial é a ancestral comum, pois a comunicação entre duas submalhas deve ser feita através da malha pai comum às duas;

- em seguida os dados do nó que se deseja transferir são copiados;

- é criada uma nova conectividade na malha para a qual se deseja transferir os nós;

- os dados do nó copiado são passados para o nó criado.

Os métodos acessórios para a obtenção da malha ancestral comum - *CommonMesh*, para alocação de um novo nó - *AllocateNewConnection*, dentre outros, são descritos a seguir.

Malha Ancestral / Pai - FatherMesh()

Toda sub-malha deve armazenar um ponteiro para a malha da qual esta é derivada. Esse ponteiro é retornado pela função *FatherMesh()*.

Figura D.16: Montagem da Matriz de Rigidez da Sub-malha - CalcStiff

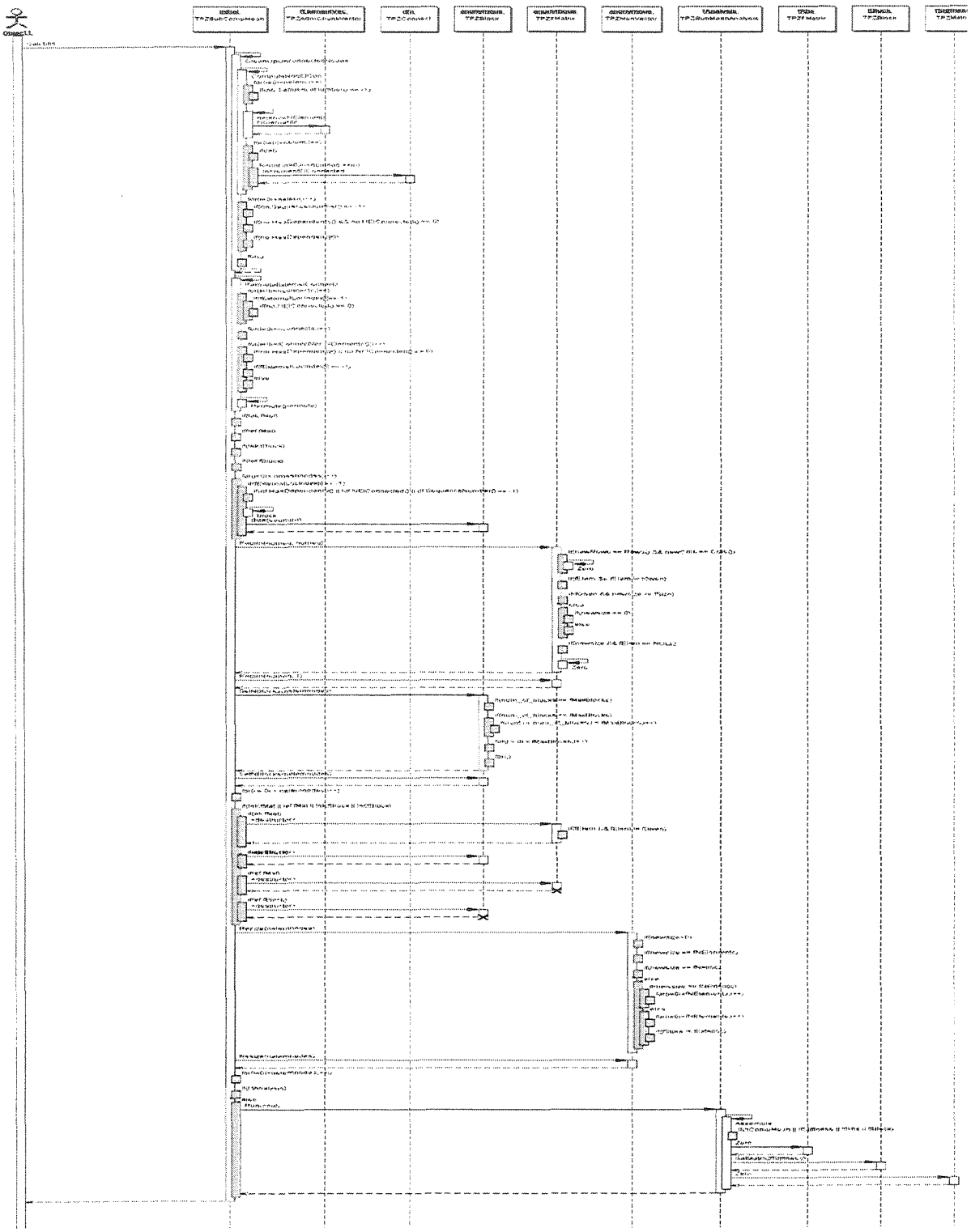


Figura D.17: Sub-malha - GetFromSuperMesh

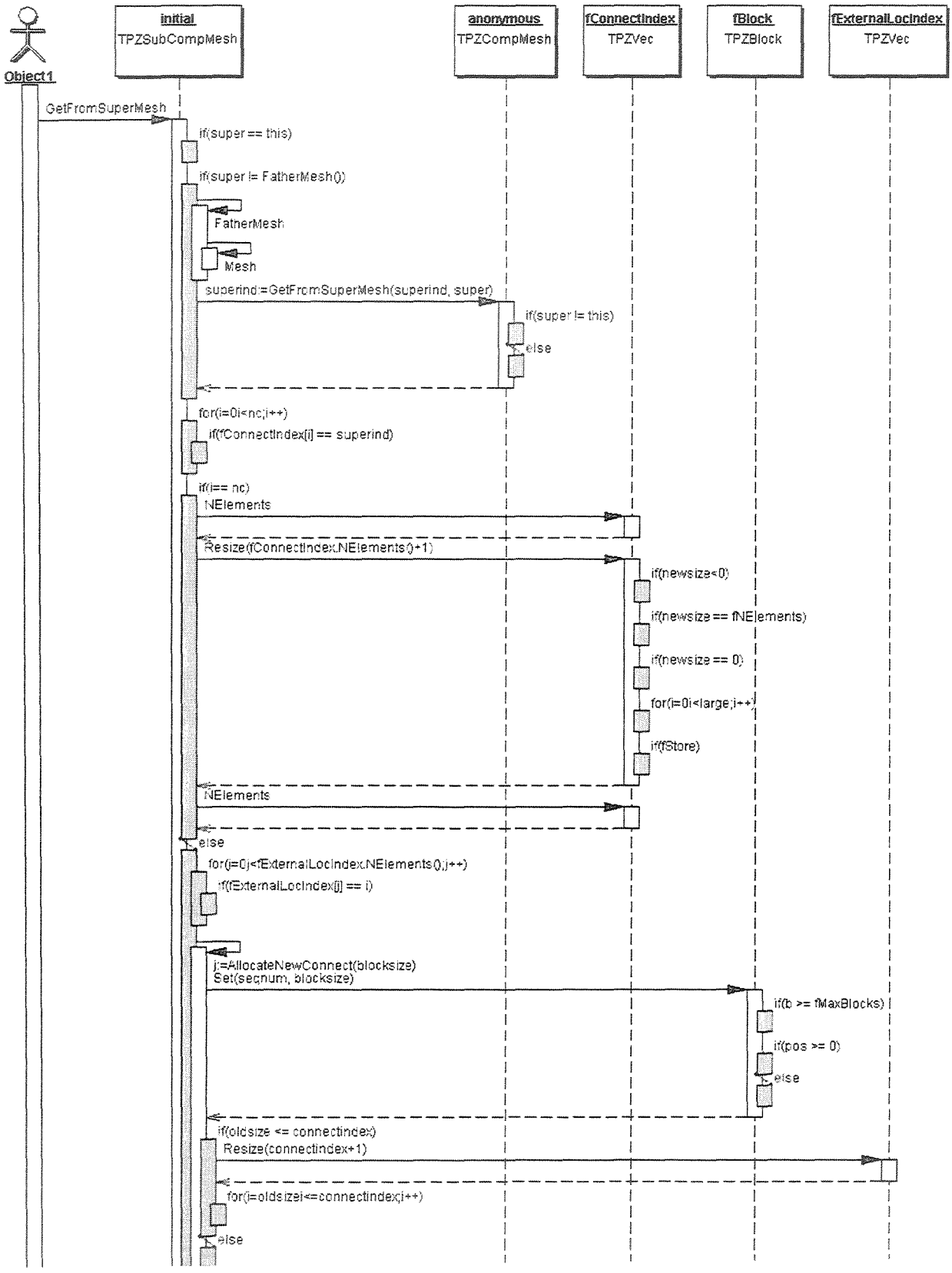
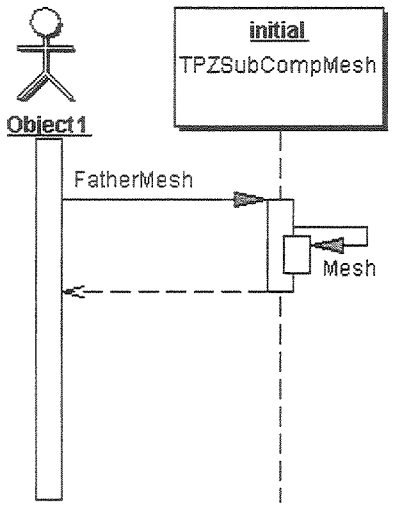


Figura D.18: Malha Pai - FatherMesh ()



O diagrama de seqüência desse método é mostrado na Figura (D.18, pág. 157).

Identificação da Malha Ancestral Comum - CommonMesh (TPZCompMesh *mesh)

Esse método, procura entre as malhas ancestrais do objeto *mesh* e do objeto corrente qual é a malha ancestral comum a ambas em nível mais baixo. Esse processo é mostrado no diagrama de seqüência da Figura (D.19, pág. 158).

Criação dos Nós na Sub-malha - AllocateNewConnect()

Conforme já mencionado anteriormente, um nó pertencente a uma sub-malha é visível para a malha pai caso ele não seja um nó interno.

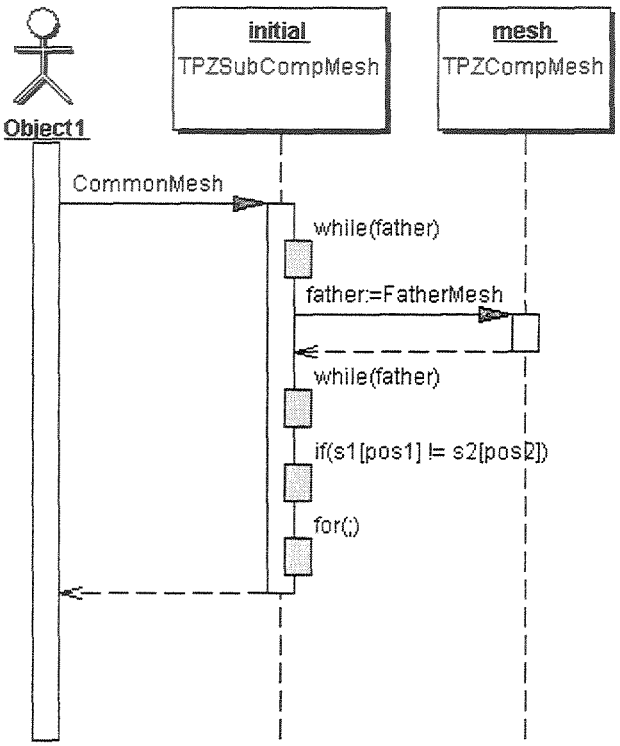
Dessa forma, a passagem de um nó da malha pai para a malha filha não é uma simples transferência.

Assim, em um primeiro momento é criada uma nova conectividade na sub-malha, para a qual serão passados ponteiros da conectividade da malha pai, tais como índices e dados relativos ao seu bloco. Tal procedimento é feito através do método *AllocateNewConnection()*, derivado da sub-malha, cujo diagrama de seqüência, onde é mostrado o fluxo de operações executadas no escopo do método é mostrado na Figura (D.20, pág. 159).

Basicamente, o diagrama citado mostra que, na alocação de uma nova conectividade são feitas as seguintes operações:

- Determinação do número de seqüência do bloco do novo nó, de modo a alocar corretamente as informações na matriz de rigidez;
- Alocação de mais uma posição no vetor de nós e do número de equações da matriz de rigidez.

Figura D.19: Identificação da Malha Ancestral Comum - CommonMesh



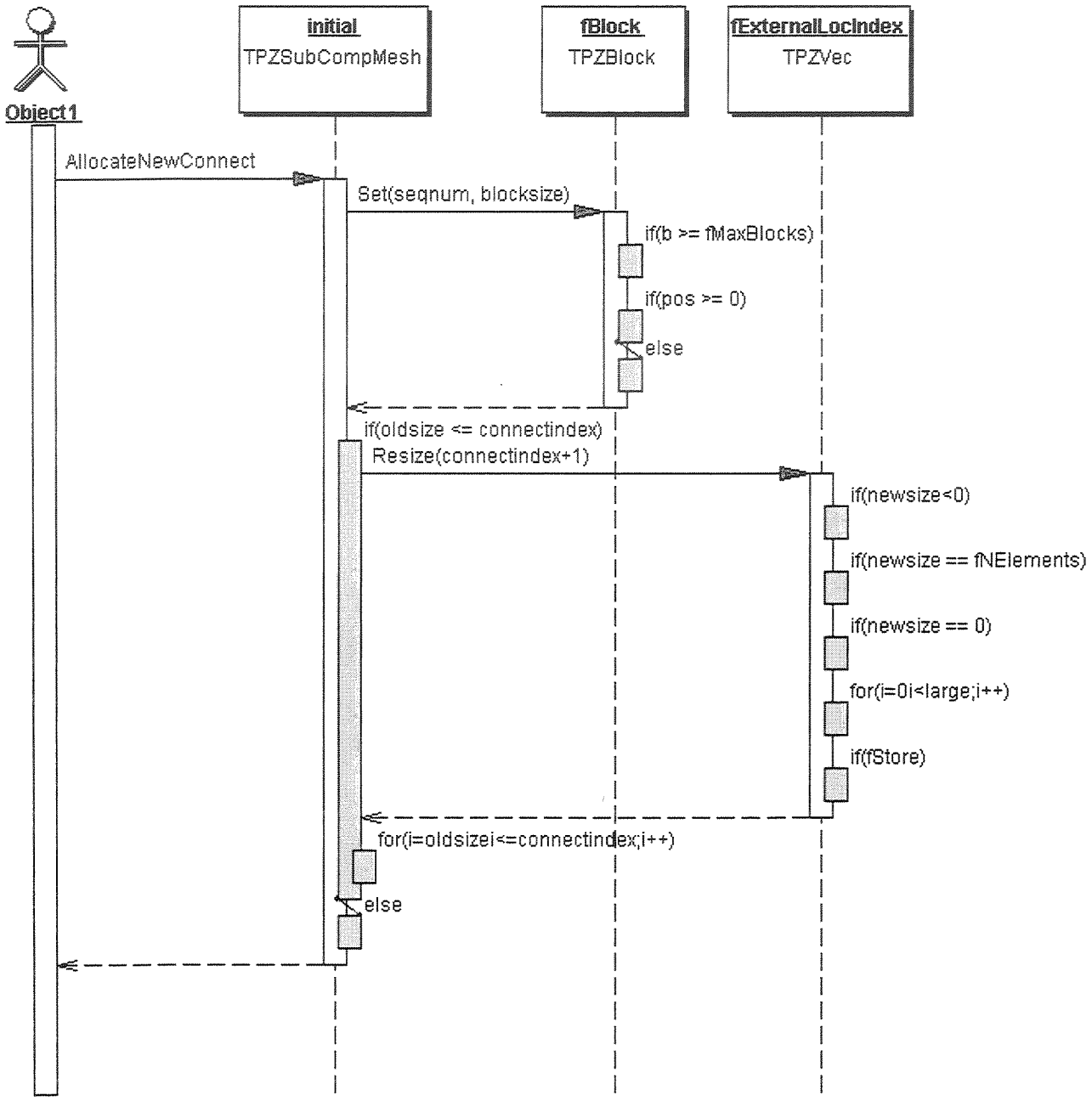


Figura D.20: Diagrama de seqüência para a alocação de uma nova conectividade.

Transferência de um elemento de uma malha pai para para uma sub-malha

A transferência de elementos entre a malha pai e sub-malha é feita através do método *TransferElement(TPZCompMesh *mesh, int elindex)*, cujo diagrama de seqüência é mostrado na Figura (D.21, pág. 161).

De fato esse método utiliza outros dois métodos privados:

- *TransferElementFrom(TPZCompMesh *mesh, int elindex)*;
- *TransferElementTo(TPZCompMesh *mesh, int elindex)*.

Assim, o procedimento básico é o seguinte: utilizando-se o método *TransferElementTo(...)*, o elemento designado é transferido do objeto atual para a malha ancestral comum, sendo dessa transferido para a sub-malha designada através da utilização do método *TransferElementFrom(...)*.

Cada um desses dois métodos tem procedimentos especiais, de modo que após a transferência não ocorram problemas com a estrutura das submalhas.

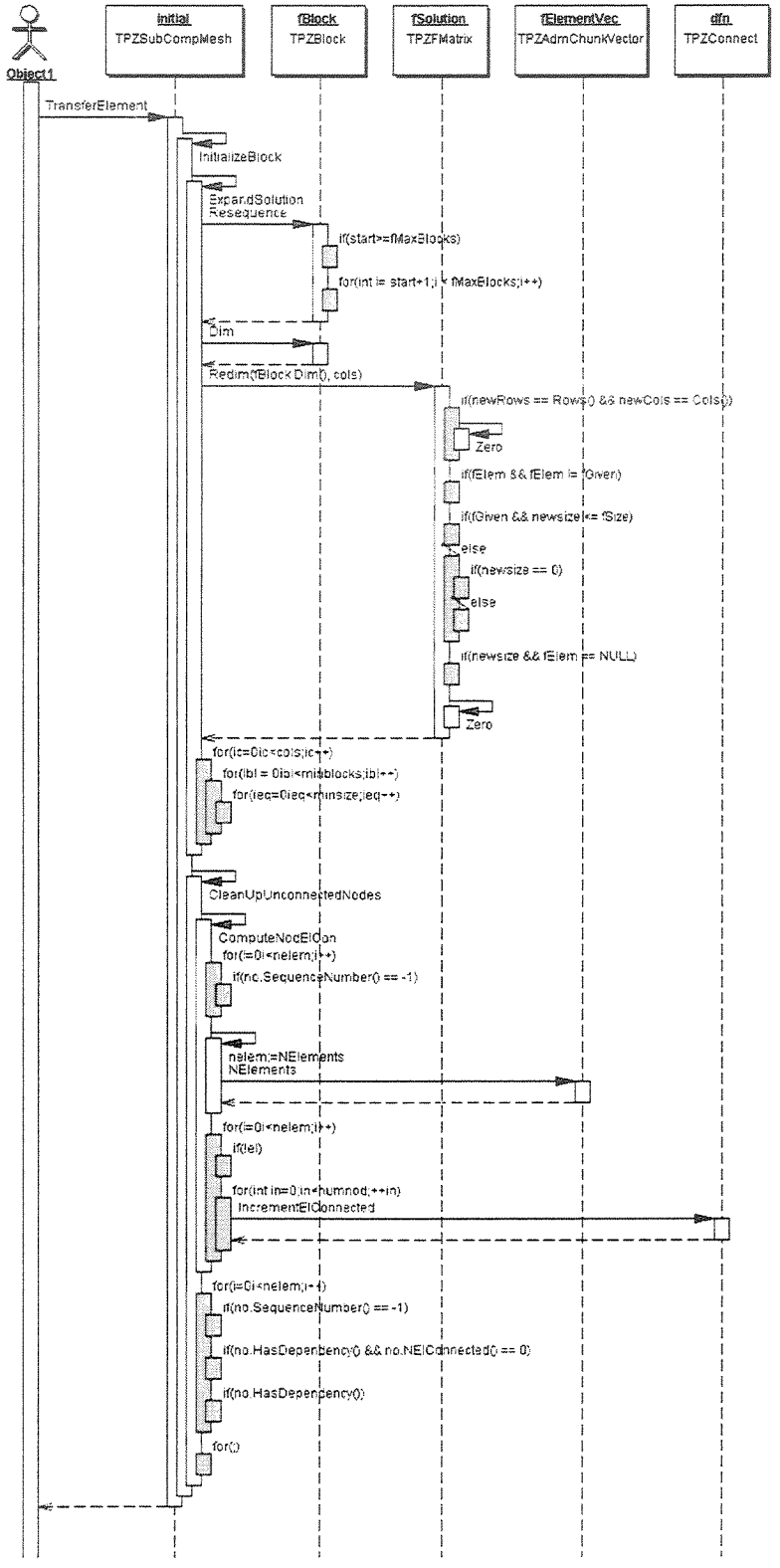
Dada a complexidade de cada um desses métodos eles são descritos separadamente na seqüência.

TransferElementFrom(TPZCompMesh *mesh, int elindex)

Esse método transfere o elemento *elindex*, para o objeto da malha *mesh* especificada, sendo realizadas as seguintes operações:

- verificação da consistência da transferência
  Foi convencionado, durante a implementação, que os elementos podem ser transferidos de uma malha filha para uma ancestral e vice versa. Assim, torna-se possível ter o controle sobre a consistência dos dados dos blocos dos nós de cada elemento.
  Dessa forma, nesse primeiro momento verifica-se se *mesh* é ancestral do objeto, caso não seja o processo é terminado com uma mensagem de erro.
- Sendo *mesh* ancestral do objeto, o elemento é transferido recursivamente para a malha pai até que essa seja a própria *mesh*;
- O processo de transferência de um elemento para a sua malha pai consiste nos seguintes passos:
- Determinação do número de nós do elemento e, para todos os nós, aplica-se rotina para tornar os nós internos externos. Nesse processo, são alocadas conectividades para os nós internos em *mesh* e definidos os seus índices, garantindo que todas as informações daquele nó serão alocadas de maneira correta na estrutura de *mesh*;
- Com as conectividades alocadas em *mesh* é criado um novo elemento, com a mesma estrutura que se encontrava na sub-malha, sendo definido o seu índice;

Figura D.21: Transferência de Elementos - TransferElement



- O elemento na sub-malha é retirado da lista de elementos, sendo feita a reordenação dessa lista.

A Figura (D.22, pág. 163) mostra a seqüência de operações descritas acima.

TransferElementTo(TPZCompMesh *mesh, int elindex)

Esse método transfere o elemento *elindex*, do objeto para a malha *mesh* especificada, sendo realizadas as seguintes operações:

- verificação da consistência da transferência: da mesma forma que para o método *TransferElementFrom*, é verificado se as malhas envolvidas tem a relação filha - ancestral.
- Sendo *mesh* descendente do objeto, o elemento é transferido recursivamente para a malha filha até que essa seja a própria *mesh* ;
- O processo de transferência de um elemento para a sua malha filha consiste nos seguintes passos:
- Determinação do número de nós do elemento e, para todos os nós, aplica-se rotina para tornar os nós internos. Nesse processo, é utilizado o método *MakeAllInternal*;
- Com as conectividades externas e internas alocadas é criado um novo elemento.

A Figura (D.23, pág. 164) mostra a seqüência de operações descritas acima.

D.5 Testes de qualificação

Durante esta implementação foram realizados testes, de modo a verificar a eficácia dos métodos implementados.

De maneira geral, cada método foi testado isoladamente durante a implementação, sendo ao final da implementação de um conjunto de tarefas, com função de modelar um determinado comportamento do objeto, foram feitos testes de integração que consistiam na comparação dos resultados obtidos na resolução de problemas com e sem a utilização dos métodos aqui implementados.

Abaixo são descritos os principais testes realizados.

D.5.1 Transferência de nós da malha computacional

O teste aqui realizado consiste na verificação da eficácia da transferência de nós entre malhas ancestrais e malhas filhas, sendo utilizados os métodos descritos acima.

Foram feitas as seguintes operações:

- transferência de nós da malha pai para sub-malha;

Figura D.22: TransferElementFrom

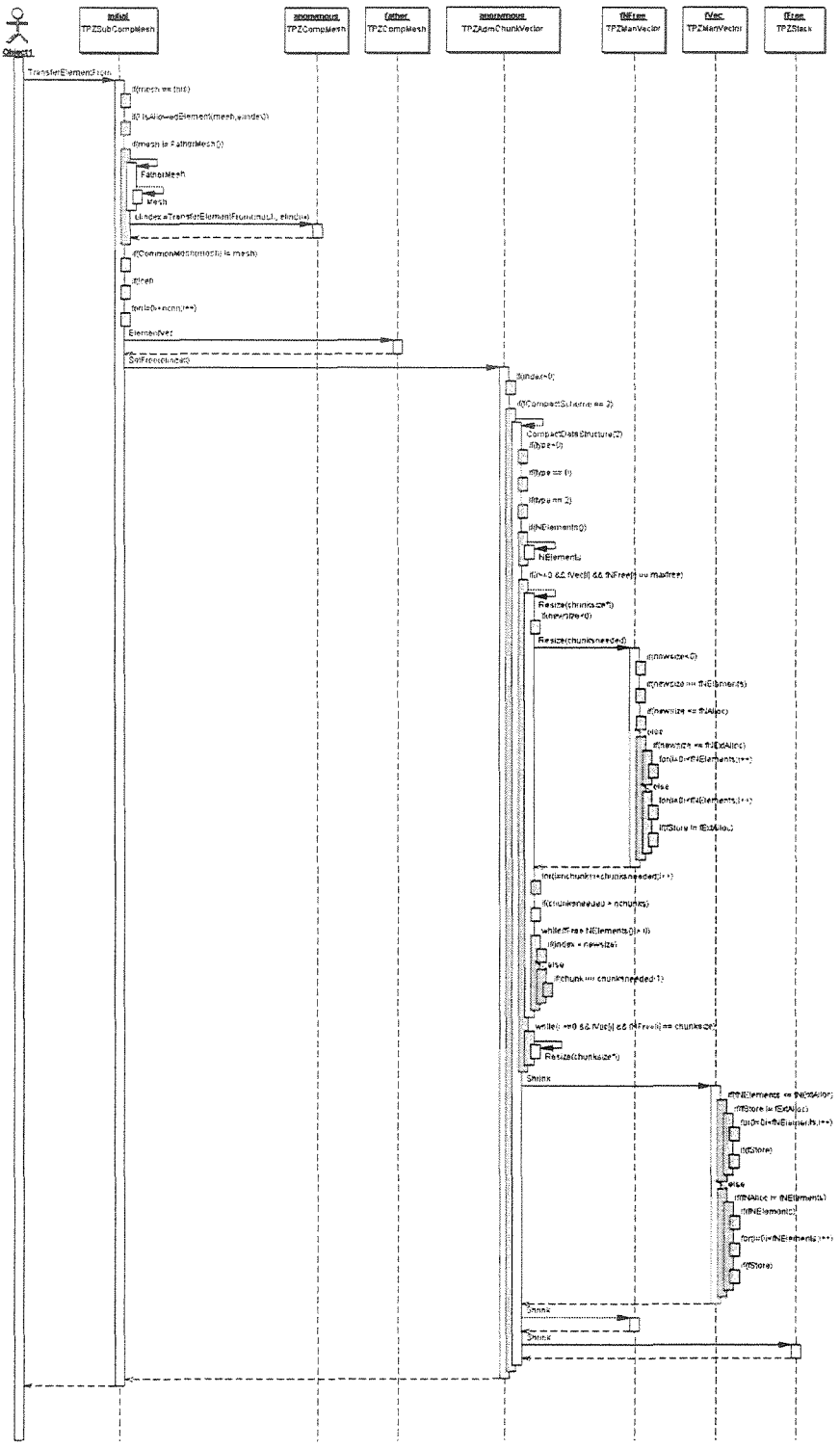
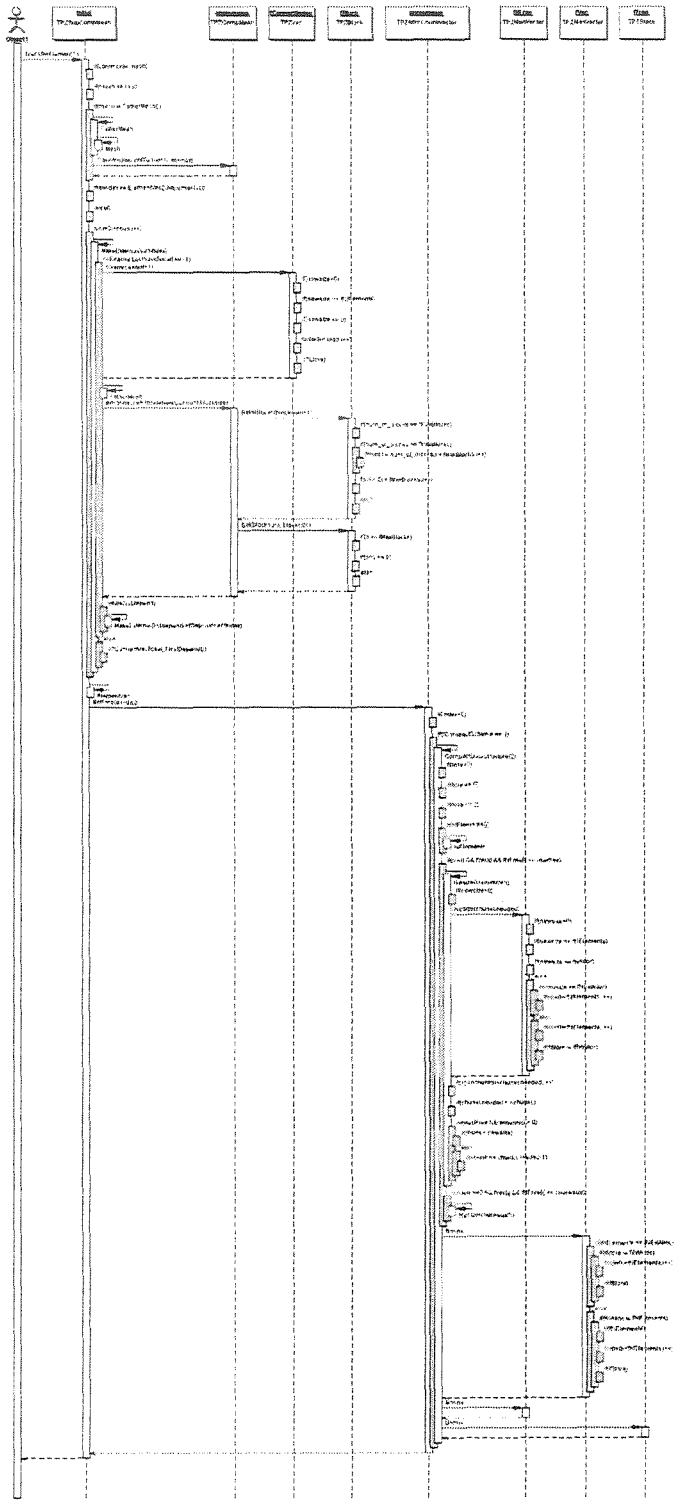


Figura D.23: TransferElementTo



- transferência de nós de sub-malha para malha ancestral;
- transferência de nós entre submalhas com ancestral comum.

Em todos os testes os resultados mostraram-se satisfatórios, pois não afetaram a matriz de rigidez global, modificando as matrizes das submalhas.

D.5.2 Transferência de elementos entre malhas

O passo seguinte foi a transferência de elementos entre submalhas.

O procedimento utilizado é aquele descrito anteriormente, sendo feitos os testes similares aos realizados para os nós.

D.5.3 Resolução de um problema de elementos finitos

A idéia adotada para a verificação da consistência dos métodos implementados é que uma mesma malha com ou sem subestruturação deve apresentar os mesmos resultados de cálculo.

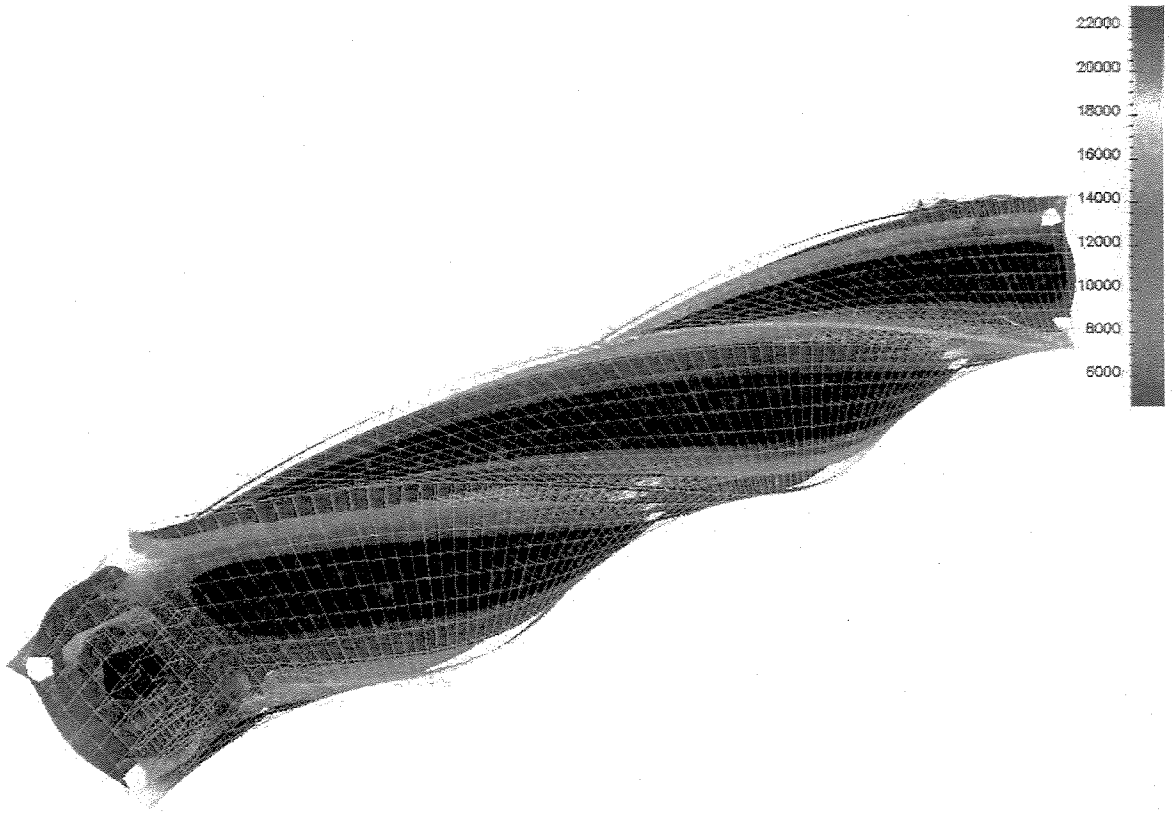
Os dados do problema são os seguintes:

- Características do material hiperelástico são as seguintes: E = 10^ 5 e ν=0.25;
- Seção quadrada de 1,0 m de lado;
- Comprimento de 10 m;
- Condições de contorno: engaste em uma extremidade e giro de 90^o na outra extremidade (sendo esse valor total aplicado em 6 passos);
- Malha composta por 10 elementos, sendo 1 elemento a cada 1 m de comprimento;

Os procedimentos realizados foram os seguintes:

- geração da malha;
- aplicação das condições de contorno;
- transformação dos elementos em submalhas;
- passagem de todos os elementos intermediários (com exceção dos dois elementos extremos) para a primeira sub-malha;
- chamada da função MakeAllInternal, de modo a transformar todas as conectividades possíveis em submalhas;
- Resolução em 6 passos, sendo a cada passo incrementado o giro na extremidade de $\frac{90^o}{6}$.

Figura D.24: Problema de Validação



Esse problema, cuja resolução já se tinha sem a utilização de subestruturação, foi comparado com o resultado obtido, sendo que esses apresentaram erros numéricos na sétima casa decimal.

A malha subestruturada apresentou resultados iguais (o erro obtido foi devido a precisão numérica - erros de truncagem) aos resultados de referência, mesmo quando utilizados procedimentos de resolução em paralelo.

A Figura (D.24) mostra o resultado da análise.

D.5.4 Conclusão

Com a implementação da classe de subestruturação de malhas e dos métodos acima descritos, criam-se perspectivas de novas abordagens para a resolução de problemas através do método dos elementos finitos, utilizando procedimentos adaptativos e computação paralela.

Tais abordagens podem vir a reduzir o custo computacional da resolução de problemas usuais, além de viabilizar a resolução, com precisão aceitável, de problemas complexos, cuja

resolução é inviabilizada devido aos requisitos de hardware para que o processamento seja feito no tempo disponível.

Referências Bibliográficas

[1] M. Ainsworth and J. T. Oden. *A Posteriori Error Estimation in Finite Element Analysis.* Pure and Applied Mathematics. First edition edition, 2000.

[2] I. Babuska and W. C. Rheinboldt. A posteriori error estimates for the finite element method. *International Journal of Numerical Methods in Engineering*, Vol. 12:pag. 1597–1615, 1978.

[3] I. Babuska, T. Strouboulis, and K. Copps. H-p optimization of finite elements aproximations: Analysis of the optimal mesh sequences in one dimension. *Computer Methods in Applied Mechanics and Engineering*, Vol. 150:89–108, 1997.

[4] E.C. Correia da Silva, P.R.B. Devloo, L. Slhessarenko, and F.A.M. Menezes. An object oriented environment for the development of parallel finite element applications. In E. Dvorkin S. Idelsohn, E. Oñate, editor, *Computational Mechanics, New Trends and Applications.* Fourth World Congress on Computational Mechanics (IV WCCM), Buenos Aires, Argentina, june 1998.

[5] B. F. de Veubeke. Displacement and equilibrium models in the finite element method, 1965.

[6] L. Demkowicz. 2d hp-adaptive finite element package (2dhp90) version 2.0. TICAM - Report 02-06, TICAM - Texas Institute for Computational and Applied Mathematics - The University of Texas at Austin, Austin, TX 78712, 2002.

[7] L. Demkowicz, D. Pardo, and W. Rachowicz. 3d hp-adaptive finite element package (3dhp90) version 2.0. TICAM - Report 02-24, TICAM - Texas Institute for Computational and Applied Mathematics - The University of Texas at Austin, Austin, TX 78712, 2002.

[8] L. Demkowicz, W. Rachowicz, and Ph. Devloo. A fully automatic hp-adaptivity. TICAM Report 01-28, TICAM - University of Texas at Austin, 2001.

[9] BRAVO Cedric M. A. Devloo, Philippe R. B. Sobre o refinamento uni-, bi- e tridimensional h-p adaptativo de elementos finitos. In *IV SIMMEC Simpósio Mineiro de Mecânica Computacional*, pages 125–139, 2000.

[10] P. R. B. Devloo. PZ : An object oriented environment for scientific programming. *Computer Methods in Applied Mechanics and Engineering*, 150:133–153, 1997.

[11] P. R. B. Devloo. Object oriented tools for scientific computing. Faculdade de Engenharia Civil - UNICAMP, 1999.

[12] P. R. B. Devloo, J. T. Oden, and P. Pattani. An hp-adaptive finite element method for the numerical simulation of compressible flow. *Computer Methods in Applied Mechanics and Engineering*, 70:203–235, 1988.

[13] Philippe R. B. Devloo. An object oriented framework for flexible mechanism simulation. In *European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS*, pages 1–13, 2000.

[14] Philippe R.B. Devloo. A three-dimensional adaptive finite element strategy. *Computers & Structures*, 38(2):121–130, 1991.

[15] P.R.B. Devloo and J.S.R.F. Alves. On the development of a finite element program based on the object oriented programming filosophy. In Ch. Hirsch, O.C. Zienkiewicz, and E. Oñate, editors, *Numerical Methods in Engineering '92*, pages 39–42, Brussels, Belgium, september 1992. First European Conference on Numerical Methods in Engineering, Elsevier.

[16] P.R.B. Devloo, F.A.M. Menezes, and E.C. Silva. OOPAR : The development of an environment for parallel computing using the object oriented programming philosophy. In B.H.V. Topping, editor, *Advances in Computational Structures Technology*, pages 151–156, 10 Saxe-Coburg Place, Edinburgh, EH3 5BR, UK, 1996. CIVIL-COMP Press.

[17] George Karypis and Vipin Kumar. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minessota, Department of Computer Science and Army High Performance Computing Research Center, Minneapolis, MN 55455, November 5 1997.

[18] P. Ladeveze and M. Zlamal. Advances in adaptive computational methods in mechanics. *SIAM J. Numer. Anal.*, Vol. 20:pag. 485–509, 1983.

[19] A. A. Novotny, J. Tomás Pereira, E. A. Fancello, and C. S. de Barcellos. An -h-p adaptive finite element mesh design strategy. Relatório de pesquisa e desenvolvimento, Laboratório Nacional de Computacão Científica - LNCC, 1999.

[20] J. T. Oden, L. Demkowicz, W. Rachowicz, and T. A. Westermann. Toward a universal h-p adaptative element strategy - part 2 a posteriori error estimation. *Computer Methods in Applied Mechanics and Engineering*, Vol. 77:pag. 113–180, 1989.

[21] John Tinsley Oden, F. Carey, Graham, and E. B. Becker. *Finite Elements - An Introdution*, volume Vol. 1. Prentice Hall Inc., New Jersey - USA, 1981.

[22] E. C. Rylo and J. L. A. de Oliveira e Souza. Desenvolvimento de interface gráfica para programas de elementos finitos. *SAE-UNICAMP*, 1996.

[23] E. C. Rylo and P. R. B. Devloo. Desenvolvimento de interface gráfica para visualizacão de superfícies 2d e 3d. *SAE-UNICAMP*, 1995.

[24] M.L. Santana and P.R.B. Devloo. Behavior of a pre-conditioner implemented using hierarchical bases. In E. Dvorkin S. Idelsohn, E. Oñate, editor, *Computational Mechanics, New Trends and Applications*. Fourth World Congress on Computational Mechanics (IV WCCM), Buenos Aires, Argentina, june 1998.

[25] Misael Luis Santana Mandujano and P. R. B. Devloo. Desenvolvimento de algoritmos de subestruturação. Master's thesis, Faculdade de Engenharia Civil - UNICAMP, March 1997.

[26] O. C. Zienkiewicz and J. Z. Zhu. A simple error estimator and adaptative for pratical engineering analysis. *International Journal for Numerical Methods in Engineering*, Vol. 24:pag. 337–357, 1987.

[27] M. Zlamal. Some superconvergence results in the finite element method. A. Dold and B. Eckmann, editores, 1975.

[28] M. Zlamal. Superconvergence and reduced integration in the finite element method. *Math. Comp.*, 32:663–685, 1978.